technische universität
dortmund

Abschlussbericht der Projektgruppe 571

Fußballspielende humanoide Roboter

Jan Hendrik Berlin, Bastian Böhm,
Sebastian Drywa, Elena Erdmann,
Michael Feininger, Florian Gürster,
Christian Kroll, Stefan Papon, Bianca Patro,
Heinrich Pettenpohl, Yuri Struszczynski

15. August 2013

Betreuer:
Dipl.-Inf. Oliver Urbann
Dr. Lars Hildebrand

Technische Universität Dortmund
Institut für Roboterforschung
http://www.irf.tu-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

Motivation

bearbeitet von: Heinrich Pettenpohl, Michael Feininger

Mit fußballspielenden humanoiden Robotern bekommt die Projektgruppe eine Gelegenheit Wissenschaft, Technik, Sport und Unterhaltung zu verknüpfen. Hier vereinen sich Bewegungsdynamik und Strategie zu einem Ereignis. Darüber hinaus besteht die Möglichkeit einen persönlichen Einblick in die autonome Robotik zu gewinnen. Durch die unterschiedlichen Aufgaben, die an die Projektgruppe gestellt werden, erhält diese einen Überblick rund um die Entwicklung eines größeren Projektes. Dazu zählen z. B. die Infrastruktur, die Kommunikation als auch die Analysemöglichkeit, auch innerhalb eines Teams.

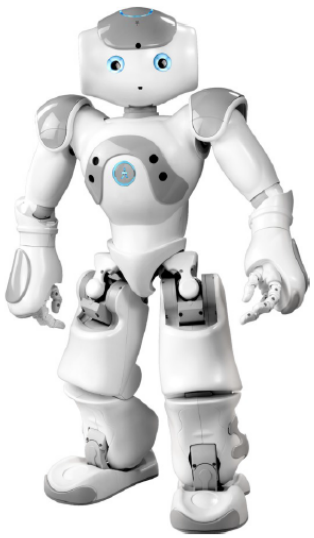


Abbildung 1.1: Nao von Aldebaran Robotics [1]

Mit den Robotern des Typ Nao (Abb. 1.1) von der Firma Aldebaran Robotics[1] werden verschiedene Turniere der Standard-Plattform-League [2] durchgeführt. Dabei werden von

[1] http://www.robocup.org

allen Teilnehmern ausschließlich baugleiche Roboter benutzt, um gegeneinander Fußball zu spielen. Die beiden größten Turniere sind die GermanOpen[2] und der internationale RoboCup[3]. Durch die standardisierte Plattform liegt der Fokus der Liga auf den verwendete Algorithmen. So kommen die Ergebnisse der Forschung in der Praxis zum Einsatz und können direkt mit anderen Teilnehmern ausgetauscht werden. Dies ermöglicht einen noch besseren Einblick in die Thematik und das Erreichen besserer Resultate.

1.1 Ziele

Die Ziele der Projektgruppe sind vielfältig. Ein Ziel ist die Entwicklung eines Debugging-Tools zur Analyse des Verhaltens der Roboter, um etwaige Fehler ausfindig zu machen. Neben der Entwicklung des Tools ist die Entwicklung zwei neuer Verhalten ein weiteres Ziel. Diese sind zum einen erforderlich um das Tool zu testen und zum anderen nutzen sie die Vorteile des Tools direkt, während der Entwicklungsphase.

Ein weiteres Ziel ist die kontinuierliche Integration mit der die Software kompiliert und getestet wird. Daraus ergibt sich als weiteres Ziel die vorhanden Module in ein neues Framework zu portieren, da das alte Framework einen veralteten und unzureichenden Simulator besitzt.

Darüber hinaus ergeben sich weitere Ziele. Beispielsweise soll die Potentialfeldmethode in die Pfadplanung eingesetzt werden. Außerdem soll ein Programm für den vereinfachten Zugriff auf den Roboter entwickelt, sowie ergänzende Interessengebiete abgedeckt werden.

1.2 Aufbau des Berichts

Der Bericht setzt sich aus fünf Teilen zusammen.

Im ersten Teil geht es um das Debugging-Tool, wobei zunächst das Logging von Daten auf dem Nao beschrieben wird. Diese Daten werden mit Hilfe des von der Projektgruppe neu entwickelten DebugTools analysiert, welches im zweiten Abschnitt des Kapitels vorgestellt wird.

Der zweite Teil beschäftigt sich mit der Infrastruktur. Es ist unterteilt in die Kapitel Integration, Framework und NaoDeployer. Bei der kontinuierlichen Integration geht es um das automatische Kompilieren und Ausführen der Software auf einem Server, sodass Fehler schnell und unabhängig gefunden werden. Dabei besteht die Software aus einem zusammenhängenden Framework. Dieses wurde von der Projektgruppe aktualisiert, was im nächsten Kapitel beschrieben wird. Unabhängig davon wurde innerhalb der Projektgruppe der NaoDeployer entwickelt. Dieses Programm vereinfacht das Übertragen der kompilierten Software auf den Nao.

[2] http://www.robocupgermanopen.de
[3] http://www.robocup.org

Im dritten Teil wird das Verhalten der Naos erläutert. Es ist unterteilt in die Bereiche Pfadplanung, Kopfwinkel, Predicted Ball, Taktiken, Torwartverhalten und Elfmeterverhalten. Bei der Pfadplanung wird der Pfad eines Roboters zu einer bestimmten Position auf dem Feld mit Hilfe der Potentialfeldmethode berechnet. In den nächsten beiden Kapiteln, Kopfwinkel und Predicted Ball, werden zwei Probleme untersucht, welche zu Beginn der Projektgruppe bei der Ballerkennung festgestellt wurden. Das Kapitel Taktiken beschäftigt sich anschließend mit der grundsätzlichen Spieltaktik auf dem Feld. Diese beinhalten hauptsächlich die unterschiedlichen Positionierungmöglichkeiten der Roboter auf dem Feld und das damit einhergehende Verhalten. Zusätzlich ist ein separates Torwartverhalten entstanden, welches im nächsten Kapitel beschrieben wird. Sollte ein Entscheidungsspiel mit einem Unentschieden beendet werden, kommt das spezielle Elfmeterverhalten zum Einsatz, welches im letzten Kapitel dieses Teils erklärt wird.

Im vierten Teil dieses Berichts erfolgt eine Aufbereitung von Themen, welche relevant für den RoboCup sind. Dazu gehören unter anderem die Challenges, sowie die Kalibrierung der Roboter. Die Challenges beschreiben kurz vor dem RoboCup veröffentlichte Aufgaben, an denen die Teams während des RoboCups teilnehmen können. Die Kalibrierung der Roboter wird während des RoboCups mehrfach durchgeführt, damit diese optimal auf jedes Spiel vorbereitet sind.

Der fünfte und letzte Teil beinhaltet das Fazit zu diesem Bericht.

Teil I

Debugging-Tool

Das Debugging-Tool mit dem Programmnamen DebugTool ist für das Finden von Fehlern im Verhalten eines Nao-Roboters konzipiert. Das Verhalten ist in der Sprache *XABSL* bzw. im aktualisierten Framework in *CABSL* geschrieben. Beide Sprachen bildet das Verhalten als einen hierarchischen Zustandsautomaten ab. Deswegen zeigt das DebugTool zum einen den entsprechenden Zustandpfad an, in dem sich das Verhalten zum eingestellten Zeitpunkt befindet, und zum anderen alle Werte der Symbole, die für die meisten Zustandsübergänge zuständig sind. Diese Symbole, sowie der Zustandspfad werden auf dem Nao-Roboter abgespeichert. Dies geschieht meistens in einem Intervall von 33 Millisekunden. Die Daten (Log) werden nach einem Spiel vom Roboter geladen und anschließend im DebugTool analysiert. Das DebugTool kann dabei mehrere Logdateien synchronisieren und abspielen. Um das entsprechende Verhalten besser zu analysieren, kann das DebugTool zusätzlich mit Hilfe eines VLC Players aufgenommene Videos abspielen. Diese können auch mit Hilfe von Offsets mit den Logdateien synchronisiert werden.

Im Folgenden wird erst auf das Loggen der Daten auf dem Nao eingegangen. Danach wird das Darstellen der Daten in dem javabasiertem DebugTool erläutert.

Kapitel 2

Logging

bearbeitet von: Yuri Struszczynski

Live Debugging während der Spiele oder Tests ist häufig sehr mühsam und oft auch völlig unmöglich. Problematisch ist es vor allem, Werte und Zustände, die nur für den Bruchteil einer Sekunde vorliegen, auf Ihre Plausibilität für das aktuell gewünschte Verhalten des Roboters zu prüfen. Eine temporäre Betrachtung des gegenwärtigen Zustands des Roboters ist daher ungenügend um die Ursache für die Vielzahl der auftretenden Probleme zu identifizieren. Schon in den letzten Jahren unserer Forschung hat sich eine Speicherung der Werte und Zustände des NAOs bis nach dem Spiel als nützlich erwiesen. Dabei werden in der aktuellsten Version unseres Frameworks unter anderem Konstanten und Symbole geloggt. Weiterhin wird der *XABSL*-Tree bzw. der neue *CABSL* -Tree, welcher Auskunft über den aktuellen Zustand im hierarchischen Zustandsautomaten gibt, aufgezeichnet.

Was wird geloggt Im Folgenden wird aufgezeigt, welche Daten gespeichert werden. Außerdem werden für alle Arten kleine Beispiele angegeben, um die Vorgehensweise zu veranschaulichen. Zusätzlich zu den Konstanten, Symbolen und den Daten des Trees können visuelle Ausgaben des Roboters aufgezeichnet werden, beispielsweise welche LEDs zum aktuellen Zeitpunkt mit welcher Farbe leuchten. Bei der Portierung des alten Frameworks wurden nicht alle Konstanten übernommen. An dieser Stelle müssen noch weitere Absprachen stattfinden, welche Werte in Zukunft überhaupt noch von Interesse sind.

Konstanten

- `field.own_ground_line.x:-3000;`
- `constants.opt_distance_to_ball_x:170;`
- `constants.max_distance_to_ball_for_sidekick_x:180;`

Symbole

- `ball.field_coordinates.x<double>;`

- `ball.distance<double>;`
- `tactic.opt_goalkick_angle<double>;`

XABSL

- `xabslTree<tree>;`
- `motion<string>;`

Nao

- `LEDs`

Skizzierung einer LogDatei Geloggt werden die Datensätze in einfachen Textdateien, sogenannte Comma-separated value Dateien (CSV), welche die Dateiendung „.xlg“ tragen. Wie man Abbildung 2.1 entnehmen kann, gibt die erste Zeile die im System vorliegenden Konstanten an, wie beispielsweise die Feld-Dimensionen. Die zweite Zeile definiert den Kopf für die eigentliche CSV Datei, die durch diesen Aufbau zur weiteren Verarbeitung schnell in eine Tabelle oder ein Array überführt werden kann. Dabei beinhalten alle folgenden Zeilen die Werte für einen definierten Zeitpunkt. Dieser Zeitpunkt wird in der ersten Spalte festgelegt, womit genau abgefragt werden kann, wann welche Zeile erzeugt wurde, sowie welche Werte zu diesem Zeitpunkt vorgelegen haben. Je nachdem wie lange der Roboter pro Update-Zyklus des `BehaviorControls` benötigt, unterscheidet sich das Intervall der Zeitpunkte, zu denen geloggt wird. In der Regel wird ca. alle 33 Millisekunden eine neue Zeile in die Logdatei geschrieben, wobei der exakte Zeitpunkt immer etwas variiert.

```
1  constants.pi:3.14159;constants.max_head_pan:114.592;constants.min_head_pan:-114.592;
2  TimeStamp<double>;motion.type<specialAction,walk,kick>;motion.walk_request.walk_type
3  -0;0;0;0;0;0;1;28;0;28;0;0;0;0;0;0;0;0;0;0;1;-1000;-2000;0;0;0;0;0;0;0;1;0;0;1e+006;0;
4  0.0106213;0;0;0;0;0;1;35;0;28;0;0;0;0;0;0;0;0;0;1;-1000;-2000;0;0;0;0;0;0;0;1;0;0;1e
5  1.38027;0;0;0;0;0;1;35;0;28;0;0;0;0;0;0;0;0;0;1;-1000;-2000;0;0;0;0;0;0;0;1;0;0;1e+0
6  1.3978;0;0;0;0;0;1;35;0;28;0;0;0;0;0;0;0;0;0;1;-1000;-2000;0;0;0;0;0;0;0;1;0;0;1e+00
7  2.47975;0;0;0;0;0;1;35;0;28;0;0;0;0;0;0;0;0;0;1;-1000;-2000;0;0;0;0;0;0;0;1;0;0;1e+0
8  2.5436;0;0;0;0;0;1;35;0;28;0;0;0;0;0;0;0;0;0;1;-1000;-2000;0;0;0;0;0;0;0;1;0;0;1e+00
9  2.58014;0;0;0;0;0;1;35;0;28;0;0;0;0;0;0;0;0;0;1;-1000;-2000;0;0;0;0;0;0;0;1;0;0;1e+0
10 2.91326;0;0;0;0;0;1;35;0;28;0;0;0;0;0;0;0;0;0;0;0;0;1200;1100;1200;1100;0;1627.88;42
11 2.98565;0;0;0;0;0;1;35;0;28;0;0;0;0;0;0;0;0;0;0;0;-4.54747e-013;1048.51;1100.07;1048
12 3.07302;0;0;0;0;0;1;35;0;28;0;0;0;0;0;0;0;0;0;0;1.13687e-013;4.44089e-016;966.348;3.
```

Abbildung 2.1: Ausschnitt Logdatei

Generell profitiert der Nutzer von einer für den Menschen schnell interpretierbaren Aufbereitung der Daten. Dagegen spricht die höhere Dateigröße gegenüber binär codierten Dateien. Das Loggen der Daten anhand der beschriebenen Vorgehensweise hat damit Vor- und Nachteile, die im Laufe dieses Kapitels noch detaillierter beschrieben und gegeneinander abgewogen werden.

Umsetzung Das Logging ist eine zentrale Funktion der `BehaviorControl`-Klassen. Im alten Framework war es so, dass während der Initialisierung des Verhaltens zusätzlich die erste Zeile mit den Konstanten, sowie die Kopfzeile der eigentlichen Tabelle geschrieben wurde. In der aktuellsten Version werden diese Daten erst nach dem ersten Aufruf der Update-Routine geschrieben, da vorher noch nicht alle Konstanten vorliegen.

Listing 2.1: Behavior::init

```
Behavior()
{
        DebugSymbolsLog::initialize(myBehaviorConfiguration.printDebugSymbols);
        logHead = false;
}

void update(const std::vector<OptionInfos::Option>& roots)
{
        ...

        // Create DebugLog Header on first update run
    if(!logHead)
    {
      // init symbols + DebugsInfos
      debugLogStatics();

      for(std::list<Symbols*>::iterator i = symbols.begin(); i != symbols.end(); ++i)
      {
        (*i)->debugLogPrintTitle();
      }

      //titles for logfile
      debugLogHead();

      logHead = true;
    }

    ...
}
```

Im laufenden Betrieb werden mit jedem Update der `BehaviorControl` zum einen alle Symbol-Klassen dazu aufgefordert, ihre Datensätze in die Logdatei zu schreiben, zum anderen werden die verhaltensspezifischen Daten, wie beispielsweise der *CABSL* -Tree aufgezeichnet. Ebenfalls werden weitere Werte direkt ohne den Umweg über die Symbols-Klassen geloggt. Das Logging im Zusammenhang mit den Symbols-Klassen wird realisiert, indem jede Klasse über eine eigene Funktion verfügt, welche einmalig aufgerufen wird und die Kopfzeile zurückgibt. Außerdem gibt es in jeder Symbols-Klasse eine weitere Funktion, welche nach jedem Aufruf der Update-Routine des `BehaviorControls` die geforderten Daten an das Logging übergibt.

Neues Framework Im Rahmen der Umstellung auf das aktuelle Framework, musste das Logging portiert werden. Dabei wurde versucht, die grundlegende Struktur beizubehalten. Als Problematisch hat sich jedoch erwiesen, dass es kaum noch Symbols-Klassen gibt, aus denen entsprechende Werte in das Logfile geschrieben werden können. Das aktuelle Verhalten verzichtet im Zusammenhang mit CABSL einfach auf einen wesentlichen Teil der alten Symbols. An den Stellen, an denen die Symbol-Klassen oder früher beinhaltete Werte schlichtweg nicht mehr vorhanden sind, wird stattdessen ein Großteil aller Daten direkt aus der Behavior.cpp/.h ausgelesen und von dort ins Log geschrieben.

Der CABSL Tree wird in der Cabsl.h erzeugt und in einen String geschrieben, welcher von der Behavior.h nach jedem Update der Instanz geschrieben wird.

Trotz der vielen Anpassung, gerade im Zusammenhang mit den fehlenden Symbols, werden viele bewährte Methoden aus dem alten Framework wiederverwendet. In Zukunft müssen jedoch noch einige relevante Ausgaben angepasst werden und gegebenenfalls veraltete

gelöscht werden. Zuletzt hatte die Abfrage des motion.walk_request und dem damit zusammenhängenden executed_motion.walk_request fehlerhafte Ausgaben erzeugt. Es gilt daher, die bereits ins Log eingefügten Variablen abschließend auf Ihre Validität zu prüfen.

Listing 2.2: Behavior::update

```
void update(const std::vector<OptionInfos::Option>& roots)
{
        ...

        for(std::list<Symbols*>::iterator i = symbols.begin(); i != symbols.end(); ++i)
        {
                (*i)->update();
                (*i)->debugLog(); // DebugPrintOut
        }

        // log values
        debugLogValues();

        //print cabsl Tree
        if(_cabslTree.empty()) _cabslTree = "Tiefe:1#OPT|Start";
        DebugSymbolsLog::printWithoutSeperator(_cabslTree);
        _cabslTree = "";

        ...
}
```

Vor- und Nachteile sowie Analysemöglichkeiten Im Laufe der vielen absolvierten Testspiele und auch auf den offiziellen Turnieren hat sich das nachhaltige Logging auf dem System als äußerst hilfreich erwiesen. Ein bereits beschriebener Vorteil des Systems liegt in der einfachen Lesbarkeit der Logdateien für den Nutzer. Generell kann die Logdatei mit wenigen Änderungen einfach in eine gängige Tabellenkalkulation geladen und darin analysiert und auf Fehler überprüft werden. Die Analyse-Möglichkeiten des Loggings sind in erster Linie auf das DebugTool begrenzt, welches die entsprechende Auswertung vornimmt. Als Hauptanforderung gilt es daher, ein neues Werkzeug zu kreieren, welches in erster Linie die effizienten Entwicklung des Verhaltens unterstützen soll. Zum Standard-Werkzeug der Software Entwicklung mit dem Nao-Framework hat sich über Jahre hinweg der Simulator entwickelt. Das neue DebugTool sollte daher, was die Ausgaben betrifft, eine ähnliche Funktionalität wie der aktuelle Simulator aufweisen. Dafür muss der State-Tree (XABSL- oder CABSL-Tree) ein neuer, wesentlicher Bestandteil der Logdateien sein und das DebugTool sollte in der Lage sein, diesen Tree zu verarbeiten und visuell aufzubereiten.

Als problematisch erweist sich die resultierende Dateigröße der einzelnen Logs, die pro Spiel schnell an die 100 Megabyte erreichen kann. Die NAO-Plattform hat zwar generell keine Probleme damit, solche großen Datenmengen lokal zu speichern, doch die Übertragung über das Netzwerk ist bei mehreren Logs (beispielsweise fünf Spieler gleichzeitig) ein Prozess, der deutlich schneller abläuft, je kleiner die Dateien sind.

Ansätze die Logs zu verkleinern, indem die Daten binär gespeichert werden, scheitern an dem relativ großen State-Tree. Dieser muss der Einfachheit halber als String gespeichert werden, was den Speichervorteil durch das binäre Speichern der Zahlenwerte erschöpft. Die resultierende optimierte Logdatei ist nur unwesentlich kleiner als das Original. Dieser kleine Speichervorteil wiegt den enormen Aufwand für ein ausgeklügeltes Datenaustauschformat zwischen `c++` und dem in `Java` geschriebenen DebugTool nicht auf. Zusätzlich wäre das Format nicht mehr menschenlesbar, weshalb dauerhaft zwei Varianten des Dateiformats gepflegt werden müssten, um das Austauschformat manuell auf Fehler überprüfen zu können.

Da der Nao nach der Deaktivierung keine Berechnungen mehr durchführen muss, könnte man die freien Kapazitäten nutzen, um die vorliegenden CSV-Dateien direkt auf dem NAO zu komprimieren. Dabei können die Logdateien teilweise um bis zu 90% verkleinert werden. Dieses Vorgehen wird noch diskutiert.

Wie bereits einleitend angekündigt wird im folgenden Kapitel die Entwicklung des neuen DebugTools beschrieben.

Kapitel 3

DebugTool

bearbeitet von: Bastian Böhm, Heinrich Pettenpohl, Yuri Struszczynski

Nachdem die Daten auf dem Nao gespeichert und anschließend auf den Computer übertragen wurden, müssen diese sinnvoll ausgewertet werden. Dazu wurde ein auf Java basierendes DebugTool entwickelt, welches unter Windows, MacOS und Linux läuft. Voraussetzung für die Plattformunabhängigkeit ist Java in der Version 1.7 oder höher. Im folgenden Teil werden erst die Funktionen des DebugTools und anschließend die innere Struktur näher erläutert. Abschließend wird auf die Umsetzung des DebugTool unter MAC OSX eingegangen.

3.1 Erste Schritte

Ein neues Projekt anlegen Damit das DebugTool verwendet werden kann, muss nach dem Start zunächst ein neues Projekt angelegt oder ein vorhandenes Projekt geladen werden. Um ein neues Projekt anzulegen, wählt man $File \rightarrow NewProject$ aus. Nun wird man aufgefordert einen Namen einzugeben und hat danach erfolgreich ein neues Projekt angelegt. Projektdateien werden unter „*username/AppData/Roaming/Debug-Tool/*“ gespeichert. Der Vorteil von Projekten ist, dass man alle wichtigen Dateien zu einem Spiel zusammengefasst vorfindet. Nicht nur die Logdateien und Videos werden zu dem Projekt gespeichert, sondern auch die Konfiguration des DebugTools. Somit ist ein Wechsel zwischen einzelnen Projekten komfortabel möglich. Das Verwalten der Projekte ist über $File \rightarrow EditProjects$ möglich.

Dateien hinzufügen Nachdem ein Projekt angelegt wurde, kann man diesem mehrere Logdateien und Videos hinzufügen. Dazu unter $File \rightarrow OpenLog$ oder $File \rightarrow OpenVideo$ die gewünschte Logdatei oder das Video auswählen.

3.2 Oberfläche

Die Oberfläche ist mit Hilfe eines Docking Frameworks realisiert. Das heißt, dass alle Daten thematisch in Fenstern zusammengefasst wurden und diese Fenster individuell verschoben werden können (Abb. 3.1). Dabei können diese auch aus dem Programm herausgezogen werden. Dies ist gerade für mehrere Monitore geeignet, so dass das DebugTool auf einem Monitor maximiert ist und auf dem anderen Monitor zum Beispiel die FieldView angezeigt wird. Im Folgendem werden die verschiedenen Fenstertypen genauer erklärt.

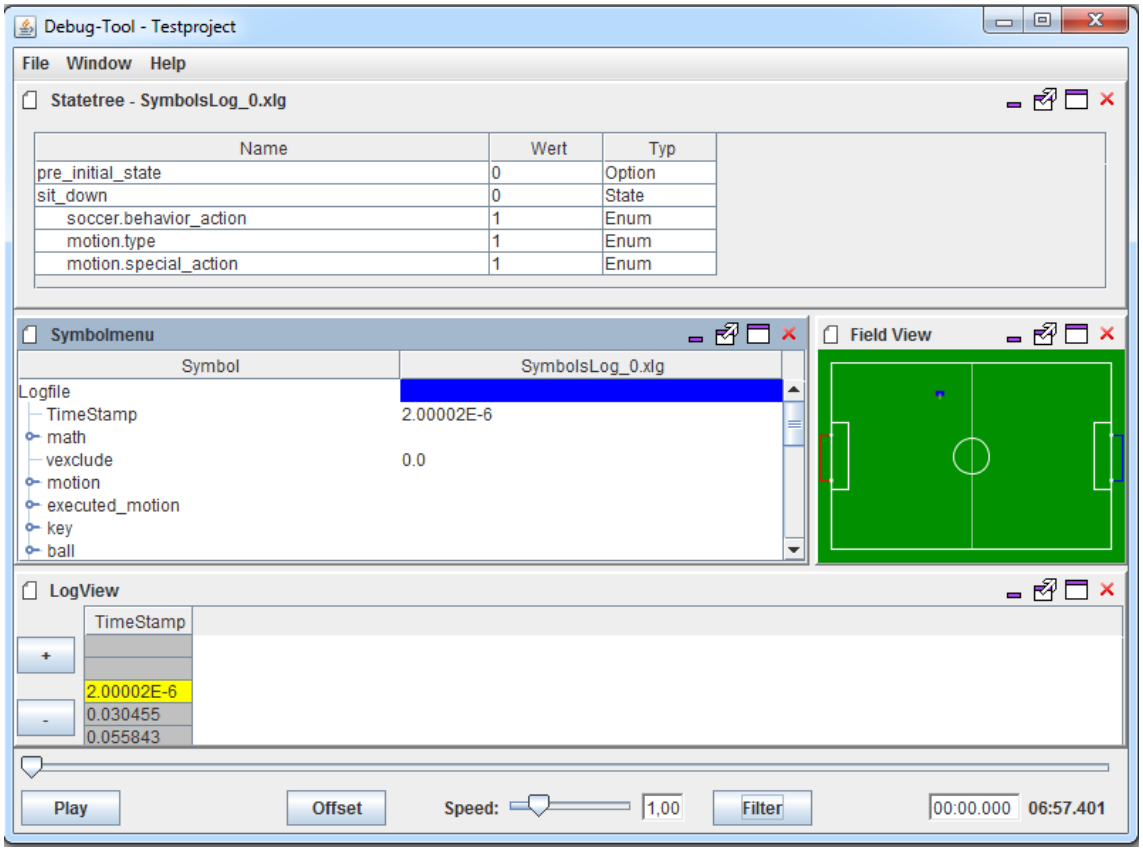


Abbildung 3.1: Die Standartansicht des DebugTools nach Laden einer Logdatei

3.2.1 Statetree

Hier wird angezeigt in welchen *XABSL*-Zuständen sich der Robotor in jedem Frame befindet. Die Darstellung ist der im Simulator ähnlich, damit man nicht jedes Mal umdenken muss, wenn zwischen Simulator und Tool gewechselt wird. Bei der Implementierung werden die Daten der Logdatei in ein Java TableModel überführt. Dies geschieht in der Klasse `TreeMenuTableModel`, welche von der Java Klasse `AbstractTableModel` erbt. Das so entstandene TableModel wird dann in der Klasse `PnlTreeMenu` verwendet, um ein neues `JTable` Objekt zu erzeugen und darzustellen.

3.2.2 Symbolsmenu

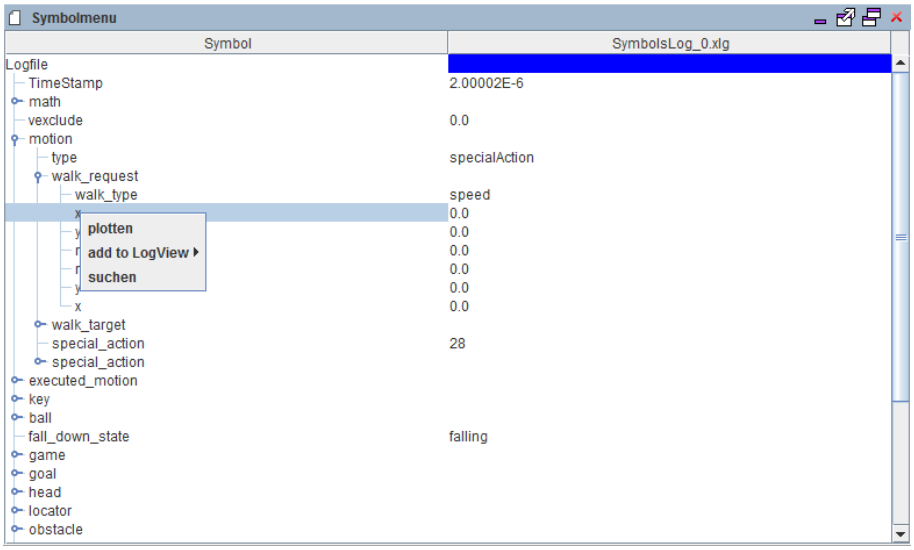


Abbildung 3.2: Das Kontextmenü des Symbolsmenu

Im Symbolsmenu werden alle, in der Logdatei vorkommenden Symbole, als Baum dargestellt (Abb. 3.2). Zusätzlich wird in der ersten Zeile die Farbe angezeigt, die dieser Logdatei und auch dem Roboter in der FieldView-Ansicht vom DebugTool zugeordnet wurde. Wird das Ende einer Logdatei erreicht, färbt sich die entsprechende Spalte im SymbolsMenu rot.

Ein Problem bei der Umsetzung ist, dass in Java keine Swing Komponente existiert, welche die Funktionalität einer Baumstruktur mit der einer Tabelle verbinden. Aus diesem Grund greift das Symbolsmenu auf ein TreeTable[1] Grundgerüst zurück, welches den `JTree` mit der `JTable` verbindet. Um die Symbole der Logdatei in der Baumstruktur darzustellen werden zunächst die einzelnen Symbole, welche eine Form wie *name.name.name* haben, in ihre einzelnen Bestandteile aufgetrennt. Das bedeutet, dass aus solch einem Block von drei Symbolnamen die mit Punkten getrennt sind, ein Java `List` Objekt gemacht wird, welches drei Namen enthält. Die so entstandenen einzelnen Symbolnamen werden dann als Eltern oder Kindknoten dem `JTree` hinzugefügt. Falls, im bis dahin schon erzeugten Baum, Symbolnamen doppelt vorkommen, werden diese nicht hinzugefügt.

Ein Kontextmenü ist über einen Rechtsklick erreichbar und beinhaltet die folgenden beschriebenen Optionen.

Liniendiagramme Beim Auswählen von „plotten“ wird das ausgewählte Symbol graphisch als Liniendiagramm aufbereitet (Abb. 3.3). Dabei werden die gleichen Symbole aus verschiedenen Logfiles automatisch in einem Diagramm zusammengefasst, wobei es Checkboxen am linken Rand ermöglichen, einzelne Logdateien auszufiltern. Zusätzlich ist das Diagram über das Ziehen und Loslassen der linken Maustaste zoombar. Die Plus- und Minus-Buttons am linken Rand ermöglichen es außerdem, den Wertebereich zu skalieren.

[1] http://hameister.org/JavaSwingTreeTable.html

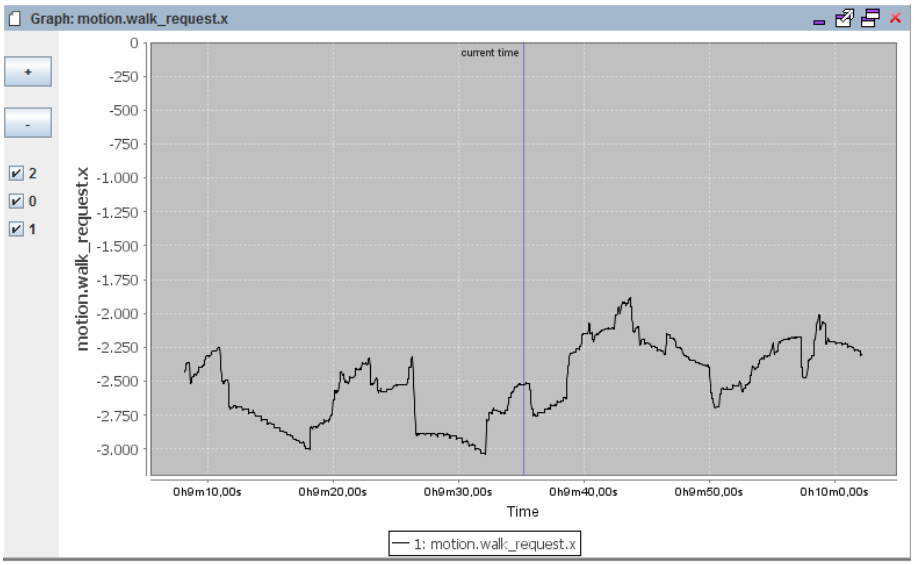


Abbildung 3.3: Graphische Darstellung mehrerer Symbols

Die Diagramme sind mit der Bibliothek JFreeChart enstanden, welche sich um die korrekte Darstellung der gelieferten Daten kümmert.

add to LogView Beim Auswählen von „add to LogView“ wird ein Symbol dem LogView Fenster hinzugefügt. siehe → *LogView*

suchen Der Eintrag „suchen“ ist noch ohne Funktion

3.2.3 FieldView

Die FieldView dient dazu, sich den Spielverlauf nochmal darstellen zu lassen (Abb. 3.4). Jeder Roboter wird mit einer, zum Symbolsmenu passenden, Farbe gekennzeichnet. Hinzu kommt, dass für jeden Roboter die Orientierung, durch eine gelbe Linie, eingezeichnet wird. Für jeden Roboter wird die von ihm „gedachte“ Position des Balls eingezeichnet. Der Ball bekommt eine rote Umrandung, die immer intensiver wird, je länger der Roboter den Ball nicht gesehen hat. Als zusätzliches Feature kann man sich den Bewegungsverlauf darstellen lassen.

Für die graphische Darstellung wird `java.awt.Graphics` verwendet, um auf Java Komponenten zeichnen zu können. Von dem eigentlichen Feld sind die Spieler, der Ball und die Laufwege der Spieler getrennt in einer anderen Klasse realisiert. Das liegt daran, dass die Feldspieler sich anhand der Logeinträge bewegen müssen. Die Konfigurationsmöglichkeiten, welche durch das Kontextmenü erreichbar sind, werden mit Hilfe von `Java Properties` gespeichert.

Das Kontextmenü ist über einen Rechtsklick erreichbar und beinhaltet folgende Optionen.

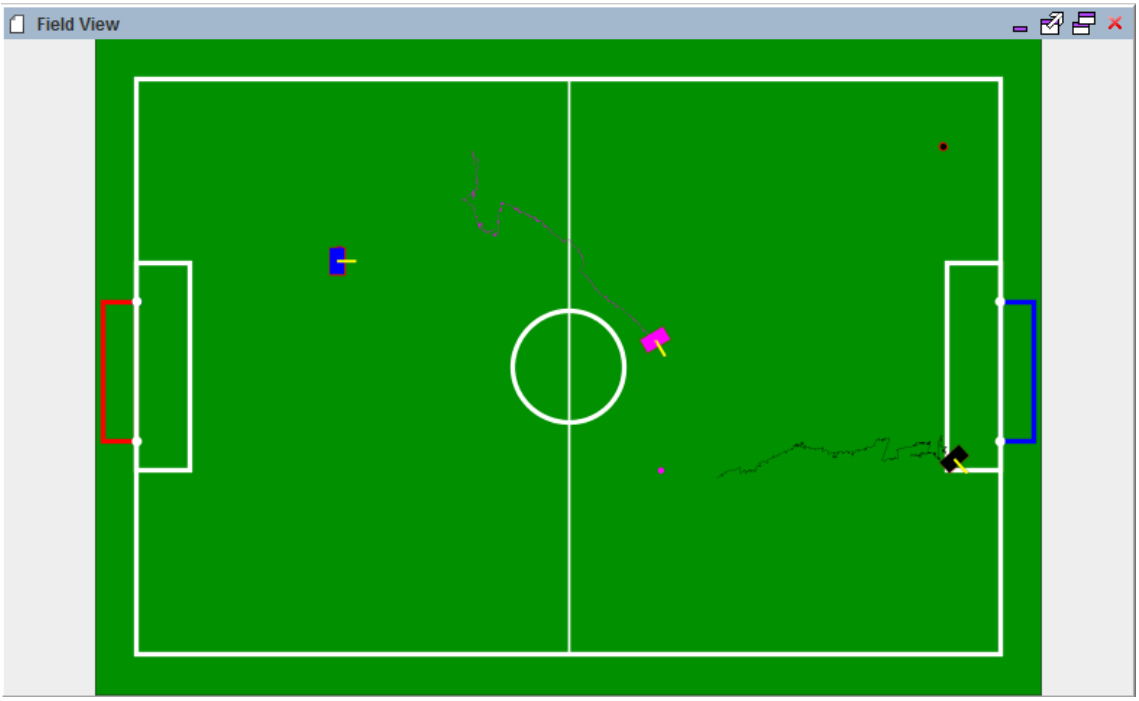


Abbildung 3.4: FieldView mit drei Robotern und Bewegungslinien

Swap Team Beim Auswählen von „Swap Team“ werden die Seiten der Teams gewechselt.

Movement Histroy on/off Beim Auswählen von „Movement Histroy on/off“ werden die Bewegungslinien aktiviert/deaktiviert.

Properties Beim Auswählen von „Properties“ werden die Eigenschaften der FieldView aufgerufen. Dort können die Länge der Bewegungslinien eingestellt und aktiviert/deaktiviert werden.

3.2.4 LogView

Die LogView ermöglicht es, zum aktuellen Zeitpunkt, direkt in die Logdatei zu schauen. Zusätzlich werden auch die Daten der vorherigen und nächsten zwei Zeitpunkte angezeigt. Möchte man den sichtbaren Bereich vergrößern, so ist dies über den Plus- und Minusbutton möglich. Um sich die Daten der Logdatei anzusehen muss man der LogView ein oder mehrere Symbols über das Symbolsmenu hinzufügen.

In der Implementierung wird die LogView durch eine `JTable` mit dazugehörigem `JTableModel` realisiert. Die Spalte mit dem aktuellen Timestamp aus der Logdatei ist in gelb eingefärbt und die Felder der Tabelle, welche die Namen der Logdateien enthalten, bekommen die selbe Farbe, die sie überall im Tool erhalten haben. So kann man auf einen Blick erkennen, welcher Eintrag im Symbolsmenu zu welchem Spieler in der FieldView und

in der LogView gehört. Realisiert wird dies mit einem selbst geschriebenen CellRenderer, welcher den vorhandenen `DefaultTableCellRenderer` überschreibt.

3.2.5 TimeLine

Die Timeline ist zentrales Steuerelement des DebugTools. Hier kann man zu bestimmten Zeitpunkten springen, das Abspielen starten und stoppen, die Abspielgeschwindigkeit einstellen und Offsets, sowie Filter konfigurieren (Abb. 3.5). In die beiden Textboxen können auch direkt Werte eingeben werden, wie zum Beispiel bei der Abspielzeit im Format „Minute:Sekunde.Millisekunde“ oder „Sekunde.Millisekunde“ .

Abbildung 3.5: Timeline Übersicht

Offset Durch verschiedene Faktoren, wie zum Beispiel das Hochfahren des Frameworks, ist es wahrscheinlich, dass nicht alle Roboter zur selben Zeit mit dem Loggen von Daten anfangen. Deshalb ist es mit der Offset Option möglich, den Spielstart (oder jeden anderen beliebigen Zeitpunkt) zeitlich für jede Log- und Videodatei festzulegen (Abb. 3.6).

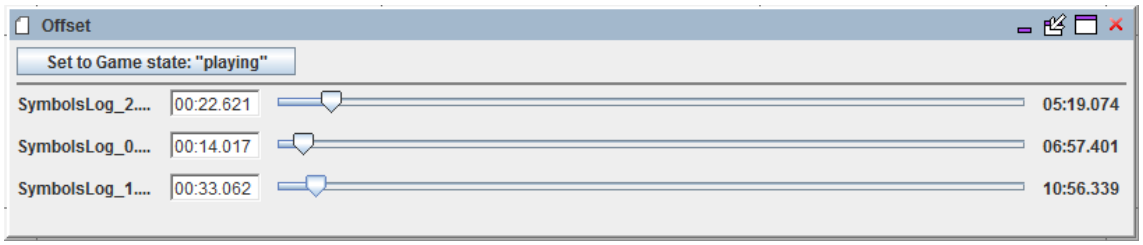


Abbildung 3.6: Offset Fenster mit drei eingestellten Offsets

Die Einstellungen werden zu dem jeweiligen Projekt gespeichert und werden automatisch geladen, wenn man das Projekt erneut öffnet.

Filter Ein Filter wird in JavaScript geschrieben und soll später dafür genutzt werden Symbole beim Erreichen bestimmter Bedingungen farbig hervorzuheben oder Markierungen auf der Zeitleiste zu setzen. Seit Java 6 ist Rhino[2] als Interpreter für JavaScript integriert.

StateTreeParser In der Vergangenheit war es schwierig herauszufinden, warum der Roboter sich in bestimmten Situationen nicht schnell genug entschieden hat. Solche langdauernden Entscheidungen können das gesamte Spiel oder spielentscheidende Situationen beeinflussen. Manchmal liegt der Grund dafür darin, dass sich der Roboter im Verhalten nicht lange genug in einem Zustand aufhält. Das heißt, er flackert oder springt zwischen

[2] https://developer.mozilla.org/en-US/docs/Rhino

zwei Zuständen hin und her und kann deshalb keine vernünftige Entscheidung treffen. Um häufig besuchte Zustände, im Verhalten Options und States genannt, zu finden, gibt es im DebugTool den StateTreeParser.

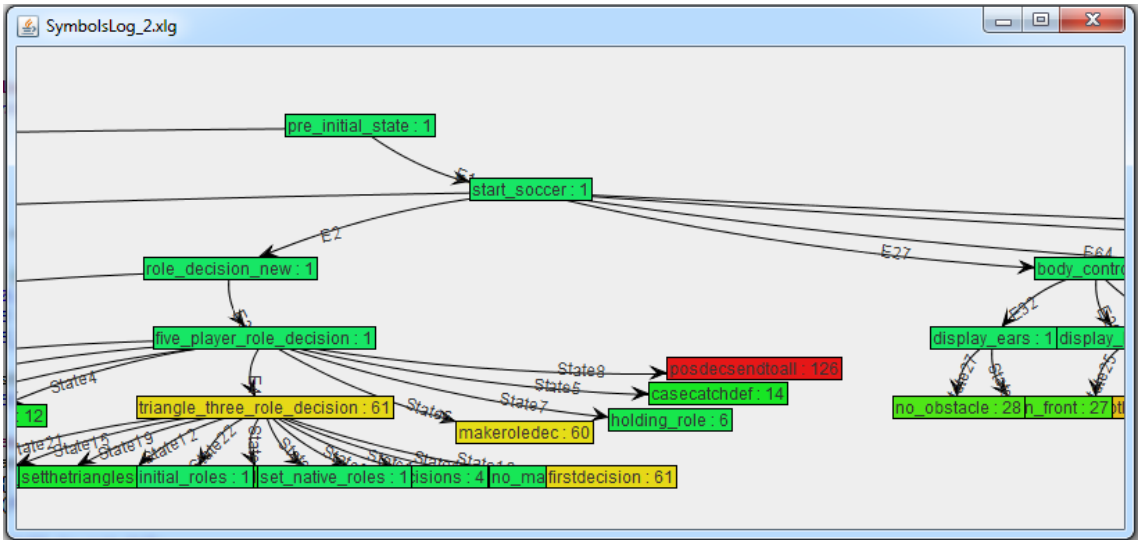


Abbildung 3.7: Ein Teil des Option-Graphs einer Logdatei

Der StateTreeParser ist eine visuelle Unterstützung bei der Analyse von Log Dateien. Er ermöglicht dem Benutzer eine Gesamtübersicht über alle Zustände, die der Roboter während des Spiels erreicht hat. Dabei werden besonders häufig besuchte Zustände farbig hervorgehoben und zusätzlich mit der Häufigkeit versehen, mit der dieser Zustand erreicht wurde. So ist es möglich häufig besuchte Zustände schnell zu identifizieren, um dadurch den Bereich des Fehlers im Verhalten einzugrenzen.

Realisiert wurde der StateTreeParser mit Hilfe des JUNG2 Frameworks, welches für die Darstellung von Graphen in Java entwickelt wurde. In der Implementierung ist das Graph<V,E> Interface das wichtigste Element im Framework. Bei der Instanziierung des Interfaces kann man auf schon implementierte Klassen zurückgreifen, die verschiedene Arten von Graphen abdecken. Beim StateTreeParser wird der DirectedSparseMultiGraph verwendet, siehe Listing 3.1.

Listing 3.1: JUNG2 Graph Objekt

```
Graph<StateTreeNode, String> g =
        new DirectedSparseMultigraph<StateTreeNode, String>();
```

Der erste Parameter steht dabei für den Datentyp der Knoten, der zweite Parameter für den der Kanten. Dem so entstandenen Graph Objekt müssen jetzt die Knoten und dazugehörigen Kanten, welche man durch das Parsen der Logdatei erhält, zugewiesen werden. Die Darstellung und Anordnung der Knoten ist, wie alles andere im JUNG2 Framework auch, frei konfigurierbar und kann bei Bedarf komplett selbst geschrieben werden. Viele mitgelieferte Implementierungen können aber auch direkt verwendet werden.

3.3 Interne Struktur

Das DebugTool besteht aus mehr als 60 Klassen, sowie aus 25 zusätzlichen Bibliotheken. Die komplette innere Struktur ist dementsprechend komplex. Deswegen wird die grundlegende Struktur am Beispiel des FieldView und TimeLine Fensters erklärt. Da Java eine objektorientierte Sprache ist, werden immer Objekte von den Klassen erstellt, welche dann, je nach Programm, miteinander interagieren. In diesem Abschnitt werden die Objekte einer Klasse als kursiver Klassenname geschrieben. Die Klassen mit dem Präfix „Pnl“ sind erweiterte JPanel Klassen, welche dann Teile der GUI darstellen.

Das DebugTool ist mit Hilfe des Beobachter-Entwurfsmusters realisiert. Dabei gibt es ein Objekt, welches als „Veröffentlicher“ handelt. Es besitzt eine Liste von Beobachter-Objekten, welche benachrichtigt werden sobald bestimmte Ereignisse eintreten. Beim DebugTool sind solche „Veröffentlicher“ z. B. der XabslReader, der XabslReaderManager oder auch die TimelineData Klasse (siehe Abbildung 3.8).

Konkret am Beispiel der FieldView und der TimeLine wird nachfolgend beschrieben wie die Logdatei in das DebugTool geladen wird und danach alle Programmteile über die geladene Logdatei informiert werden. Anschließend wird darauf eingegangen, wie die Logdatei abgespielt wird.

In das DebugTool wird eine Logdatei geladen, indem beim *XabslReaderManager* die Methode `loadXabslLog(File file)` mit dem Dateinamen aufgerufen wird. Dieser erstellt ein neues Objekt vom Typ XabslReader, welches die Validität der Logdatei prüft. Danach lädt der *XabslReader* mit Hilfe eines internen Threads die Logdatei in die Vektoren `columnVector`, `headerTypesVector` und `timeVector`. Dabei beinhaltet der `headerTypesVector` die Spaltenüberschriften, der `columnVector` die eigentlichen Daten der Spalten und der `timeVector` nur die Zeitstempel-Spalte.

Sobald die Logdatei vollständig geladen ist, ruft der *XabslReader* die Methode `addXabslReader(XabslReader xr)` beim *XabslReaderManager* auf. So kann dieser alle aktiven und vollständigen *XabslReader* im Vektor `xrVector` verwalten. Sobald dieser Vektor durch Hinzufügen oder Löschen geändert wird, werden alle Beobachter des *XabslReaderManagers* über ihre `update(Observable o, Object arg)` Methode darüber informiert. Dabei wird der Vektor als Objekt mit übergeben.

So erfährt z. B. das *PnlField* von der Existenz einer neu geladener Logdatei. Das *PnlField* ist für das Zeichnen der FieldView verantwortlich. Es zeichnet das Spielfeld und verwaltet die PnlFieldObject's im Vektor `vPnlFieldObject`. Diese sind für das Zeichnen der Roboter auf dem Spielfeld zuständig und zeichnen dabei auch den wahrgenommenen Ball, sowie ihren bisherigen zurückgelegten Weg ein. Sobald also eine neue Logdatei in einen *XabslReader* geladen wurde und der *XabslReaderManager* seine Beobachter darüber informiert, erzeugt das *PnlField* pro geladene Logdatei ein neues *PnlFieldObject*. Dieses kann über eine Referenz direkt auf den *XabslReader* zugreifen, um die benötigten Daten graphisch darzustellen. Außerdem wird das *PnlFieldObject* zum Beobachter des *XabslReaders*, um immer über den aktuellen Datensatz informiert zu werden.

Wenn nun der Benutzer bei der TimeLine das Abspielen der Logs aktiviert, wird dies an die *TimelineData* weitergegeben. Diese beinhaltet einen Thread, welcher ungefähr eine Milli-

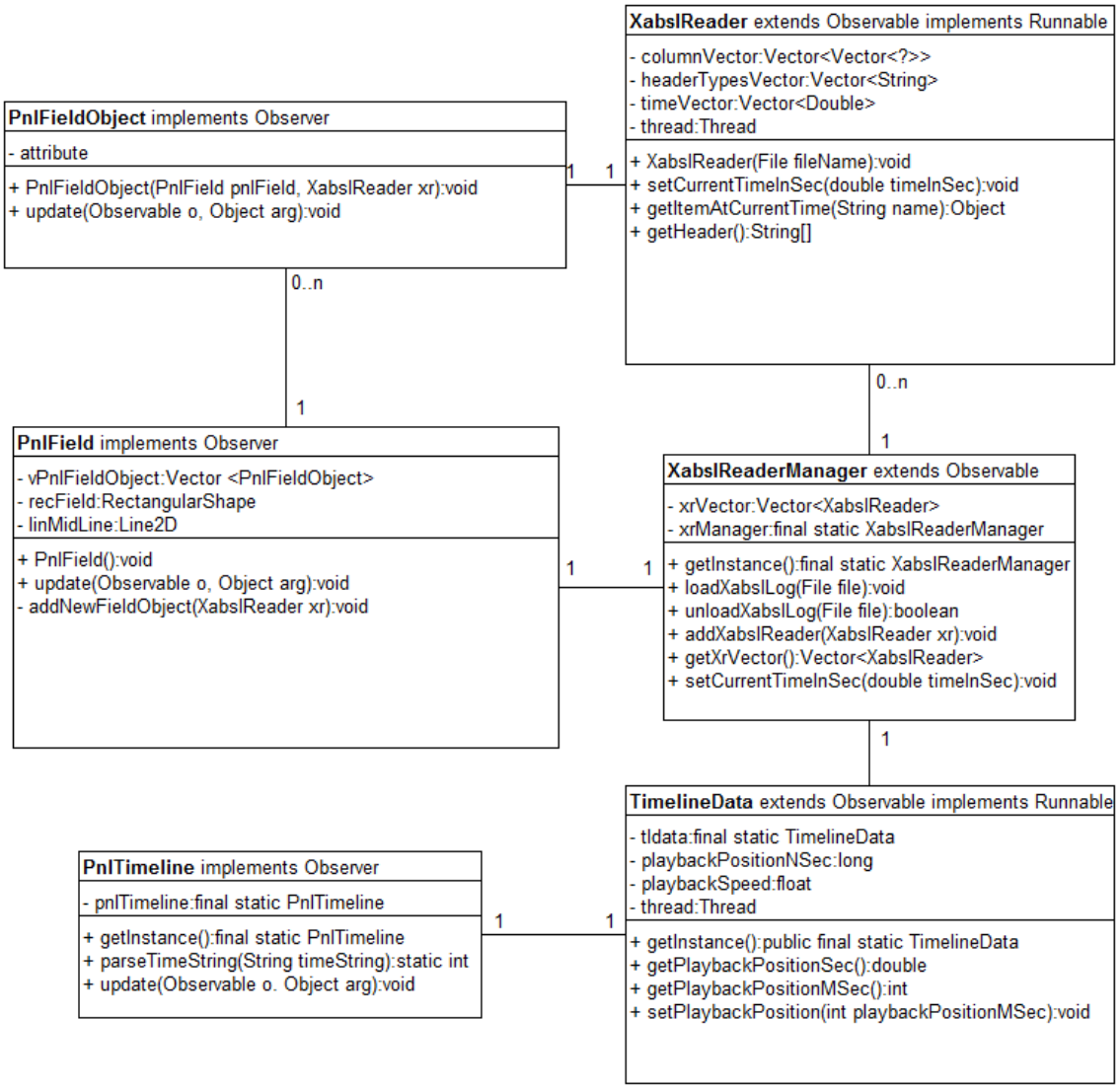


Abbildung 3.8: Skizze der internen DebugTool Struktur

sekunde lang schläft, um die Prozessorlast zu verringern und dann nanosekundengenau die Abspielposition berechnet. Dies ist je nach Abspielgeschwindigkeit unterschiedlich. Dabei ruft der Thread die Methode `setCurrentTimeInSec(double timeInSec)` des *XabslReaderManagers* auf, welcher dann die gleichnamige Methode aller *XabslReader* aufruft. Diese informieren in ihrem eigenen Thread alle Beobachter, wenn sich aufgrund der Abspielposition eine neue Zeile der Logdatei als aktuell herausstellt. Auch die *TimelineData* informiert alle ihre Beobachter über eine Änderung der Zeit. In diesem Beispiel wird das *PnlTimeline* informiert, welches dann die aktuelle Abspielposition darstellen kann.

3.4 MAC OSX

Die Umsetzung des DebugTools erfolgt nach anfänglicher Diskussion in Java. Erste Versuche mit C# waren zwar sehr erfolgreich, doch die mehrheitliche Entscheidung viel auf die Entwicklung eines plattformunabhängigen Programms in einer Java Umgebung.

Im fortgeschrittenen Entwicklungsprozess des Projektes zeigt sich jedoch, dass der Einsatz des Plugins VLCj nicht auf allen Plattformen auf dem gleichen Entwicklungsstand ist. VLCj stellt eine Komponente bereit, die es ermöglicht ein Video direkt in einem Fenster innerhalb einer graphischen Oberfläche darzustellen. Dieses Fenster kann wie alle anderen Fenster im Rahmen der GUI Verschoben, in der Größe verändern und angedockt werden. Problematisch ist jedoch, dass unter MAX OSX generell ein Fehler geworfen wird, sobald ein Video über *EmbeddedMediaPlayerComponent* eingebettet wird. Die Entwicklerforen zu VLCj konnten an dieser Stelle ebenfalls keine umfassende Lösung bieten.

3.4.1 Fehler

JAWT_GetAWT must be called after loading a JVM
Exception in thread „AWT-EventQueue-0“ java.lang.UnsatisfiedLinkError: Can’t load JAWT

Ein Ansatz besteht in der Verwendung der alternativen Komponente *DirectMediaPlayer*. Videos können hierbei auch unter MAC OSX abgespielt werden, doch diese müssen bereits über eine fest definierte Größe verfügen, was ein nachträgliches anpassen der Fenstergröße verhindert. Dieser Ansatz hat sich als nicht praktikabel erwiesen.

Es kann durchaus sein, dass in neueren Versionen von VLCj bzw. der Java OpenJDK das zugrundeliegende Problem gelöst wird. Hier sollten in Zukunft weitere Versuche unternommen werden.

3.5 Fazit und Ausblick

Das DebugTool ist soweit fertiggestellt, dass es alle Funktionalitäten bietet um damit ein Verhalten auf seine Fehler zu untersuchen. Es lassen sich Projekte mit Logdateien und Videos anlegen und zwischen ihnen wechseln. Analysen eines Fussballspiels werden durch die Darstellung der Spieler auf dem Feld, dem Blick in die Logdatei, die Übersicht über die Symbole und dem Zustandsbaum ermöglicht.

Ein Spiel der Roboter soll nicht nur durch eine Logdatei dokumentiert sein, sondern auch für den Benutzer des DebugTools auf einen Blick die wichtigsten Informationen bereitstellen. Geplant ist, dass weitere Ausbauen der Filterfunktion sowie zusätzliche Analysefunktionen wie zum Beispiel Statistiken über Zustandswechsel oder über die Wertigkeiten von Symbolen. Somit soll es das DebugTool ermöglichen, durch zum Beispiel graphische Aufbereitung von Statistiken, relevante Informationen mit nur einem Blick zu erkennen. Die Option sich ein Symbol als Graph darstellen zu lassen oder die Bewegungslinien der Roboter in der FieldView sind jetzt schon solche graphischen Zusatzinformationen, die der

Benutzer verwenden kann. Ein weiterer Punkt der Planung ist es, den StateTree auch mit mehr Informationen anzureichern als nur den reinen Daten der Logdatei. Das hilft den Benutzern schneller ein Problem zu identifizieren, um dann mehr Zeit mit dem eigentlichen Verhalten zu verbringen als mit dem Suchen nach Fehlern.

Außerdem soll der Projektgedanke noch weiter ausgebaut werden, indem auch Textnotizen und Markierungen auf der Zeitleiste editiert und gespeichert werden können.

Nachdem nun erklärt wurde wie man das Verhalten analysieren kann, werden im nächsten Abschnitt die zwei neu entwickelten Verhalten erklärt.

Teil II

Infrastruktur

Die Infrastruktur umfasst einerseits ein Betriebssystem für die Naos auf Basis von Gentoo-Linux, andererseits ein umfangreiches Framework zur Steuerung und Koordination der Roboter. Letzteres basiert zu einem Großteil auf den Forschungsarbeiten von *B-Human*, einem erfolgreichen Team der Universität Bremen und dem Deutschen Forschungszentrum für Künstliche Intelligenz (DFKI) in der RoboCup SPL. Es ermöglicht nicht nur die High-Level-Programmierung der Roboter, sondern bietet auch einen Simulator, in dem sich unter anderem komplette Testspiele simulieren lassen.

Die bis einschließlich zur German Open 2013 genutzte Version des Simulators war nicht in der Lage, ein komplettes Spiel mit zehn Spielern zu simulieren. Selbst die Simulation eines einzelnen Roboters lief deutlich langsamer ab als in Echtzeit. Als Abhilfe für dieses Problem hat B-Human im Rahmen ihres Code-Releases im Jahr 2012 [9] eine neue, deutlich schnellere Version des Simulators herausgegeben. Das Release enthält zudem ein aktualisiertes Framework, das viele Verbesserungen und Erweiterungen erfahren hat. Zusammen mit dem Simulator auf dieses neue Framework zu wechseln war daher eine naheliegende Entscheidung, auch wenn dies eine aufwändige Portierung der Dortmunder Module nach sich zog.

Im Zuge dieser Portierung war auch eine Aktualisierung des Roboter-Betriebssystems möglich. Es enthält einen neuen Kernel, der merklich stabilere WLAN- und Kamera-Treiber mitbringt. Zusätzlich aktiviert er Hyperthreading, das in Kombination mit dem neuen Framework weitere Leistungssteigerungen ermöglicht. Im alten System führte aktiviertes Hyperthreading zu massiven Stabilitätsproblemen.

Ein Ziel der Projektgruppe war die Qualität des Frameworks durch agile Techniken zu verbessern. Konkret kam hierbei die kontinuierliche Integration zum Einsatz, deren umfassende Automatisierung von Prozessen die Konzentration auf wesentliche Probleme der Softwareentwicklung ermöglicht. So wurden das Framework und das Debug-Tool automatisch beim Check-In übersetzt und beim letzterem zusätzlich Nightly-Builds angeboten.

Ein weiterer Teil der Infrastruktur ist der NaoDeployer, der die Installation des Frameworks auf die Roboter schneller und einfacher macht.

Kapitel 4

Kontinuierliche Integration

bearbeitet von: Christian Kroll

Eine funktionierende Code-Basis ist Voraussetzung, um erfolgreich in Wettbewerben wie dem *RoboCup* zu bestehen. Dabei gilt es nicht nur Herausforderungen zu meistern, die die Übersetzung eines umfangreichen Frameworks wie das von B-Human mit sich bringt. Auch die semantische Korrektheit der Algorithmen, sei es das Verhalten oder die Bildverarbeitung, entscheidet über Sieg oder Niederlage in einem Spiel. Für das Team ist es daher äußerst wichtig, Probleme in diesen Bereichen möglichst frühzeitig zu erkennen und zu beheben.

Durch Zufall auf Übersetzungsfehler zu stoßen oder permanent manuell den Ausgang von Simulationsspielen zu ermitteln, ist auf Dauer ineffizient und bindet unnötig Zeit. Besser ist es, diese Schritte zu automatisieren und maschinell auszuwerten. Eine Möglichkeit dies zu bewerkstelligen sind sogenannte *Continuous-Integration-Server*, die entscheidende Vorgaben im Entwicklungsmodell der *kontinuierlichen Integration* umsetzen.

4.1 Was ist kontinuierliche Integration?

Kontinuierliche Integration ist abstrakt betrachtet eine Sammlung von Vorgehensweisen aus dem Bereich der agilen Softwareentwicklung. Eine weithin akzeptierte Definition, an der sich auch die folgende Darstellung orientiert, findet sich in einem viel beachteten Artikel [10] des britischen Softwareentwicklers und Buchautors Martin Fowler [11].

Durch die gesamte Implementierungsphase hindurch ein lauffähiges Projekt zu gewährleisten ist das Hauptziel der kontinuierlichen Integration. Dementsprechend muss das Projekt zu jedem Zeitpunkt kompilier- und ausführbar sein. Das setzt voraus, dass die Entwickler ihre Änderungen fortlaufend in das Projekt integrieren und nicht über längere Zeit mit ihren eigenen, divergierenden Entwicklungszweigen arbeiten. Letztere bergen die Gefahr, sich nur mit großen Reibungsverlusten zu einem neuen Hauptzweig zusammenfassen zu lassen. Die automatische Übersetzung unmittelbar nach dem Einchecken gibt den Entwicklern Aufschluss über die Übersetzbarkeit ihrer Änderungen auf anderen Systemen. Dem können sich, eine erfolgreiche Übersetzung vorausgesetzt, automatische Laufzeittests anschließen, nebst Deploy auf dafür vorgesehenen Testumgebungen. Viele Probleme fallen

so unmittelbar nach Ihrer Entstehung auf und ermöglichen auf diese Weise eine frühzeitige Behebung.

Einen weiteren Schwerpunkt setzt die kontinuierliche Integration auf die Kommunikation zwischen den Projektbeteiligten. Informationen über den gegenwärtigen Zustand des Projekts sowie dessen Vergangenheit (zum Beispiel fehlgeschlagene Builds und deren Verursacher) stehen ebenso jedem zur Verfügung wie ein Download der letzten ausführbaren Version, sei es zu Test- oder Demonstrationszwecken. Martin Fowler definiert noch weitere Vorgaben, beispielsweise die Verfügbarkeit aller zum Übersetzen und Ausführen notwendigen Ressourcen in einem zentralen Repository sowie möglichst kurze Build-Zeiten, wobei er obere Schranken von zehn Sekunden nennt.

Nicht alle diese Paradigmen setzt die Projektgruppe um. Das Hauptaugenmerk liegt auf der Automatisierung des Builds des B-Human-Frameworks, um schnell auf Übersetzungsfehler reagieren zu können, die beispielsweise durch fehlende oder unvollständige Check-Ins zustande kommen. Automatisch simulierte Testspiele scheitern bisher leider am Grad der notwendigen Interaktivität des Simulators, der zwingend eine grafische Oberfläche und deren Bedienung voraussetzt. Das in dieser Projektgruppe entwickelte Debug-Tool profitiert ebenfalls von der weitreichenden Automatisierung. So sind neben Übersetzungen beim Check-In zusätzlich Nightly-Builds aktiv, bei denen ein selbsttätiger Mechanismus jede Nacht ausführbare Jar-Dateien sowie komprimierte Quellcodearchive erstellt und diese auf den Webserver der Nao-Devils-Homepage [12] kopiert.

Die technische Umsetzung der oben vorgestellten Prinzipien übernehmen *Continuous-Integration-Server* (oder kurz *CI-Server*), wobei die Projektgruppe konkret auf das Produkt *TeamCity* aus dem Hause *JetBrains* setzt. TeamCity kommt mit einer Vielzahl von verschiedenen Versionsverwaltungen, Build-Systemen und Unit-Test-Fameworks zurecht und ermöglicht so die Automatisierung von Übersetzung, maschinellen Tests und Deployment. Es überwacht ausgewählte Branches im Repository und wird umgehend aktiv, sobald Änderungen anstehen. Sogenannte *Build Agents* übersetzen letztendlich das Projekt. Sie verrichten ihre Arbeit getrennt vom CI-Server als eigenständige Programme, wobei sie idealerweise auf dedizierten Hosts laufen und mit dem CI-Server übers Netz kommunizieren. Im Fehlerfall kann TeamCity anhand des letzten Commits den Verursacher ermitteln und diesen umgehend mittels einer E-Mail oder einer Jabber-Nachricht informieren. Zusätzlich bereitet es Informationen über Projektstatus und Projektgeschichte mit verschiedenen Metriken in einer übersichtlichen Weboberfläche auf.

4.2 Einrichtung von TeamCity als CI-Server

Das System, welches der Projektgruppe für den CI-Server zur Verfügung steht, ist eine auf *Debian 6.0* basierende, virtuelle Linux-Maschine mit 32 Bit. Bevor TeamCity auf so einem System lauffähig ist, sind einige Voraussetzungen zu schaffen:

- ein Java-Development-Kit von Oracle
- Zugriff auf einen Datenbankserver wie MySQL
- Zugriff auf einen SMTP-Server

- ein Startskript gemäß *System-V-Init*

Die Installation und Konfiguration des Java-Development-Kits, des MySQL-Datenbank-Servers und des SMTP-Servers sind Gegenstand des Zwischenberichts [13]. Die folgenden Abschnitte setzten daher die Verwendung von Orcale JDK 1.7.0, MySQL 5.0 sowie Exim 4.72 voraus.

4.2.1 Erstellen einer Startumgebung

Abseits seiner Weboberfläche verrichtet TeamCity seine Arbeit größtenteils im Hintergrund. Administrative Aufgaben wie manuelles Starten oder Stoppen des Dienstes sollen so wenig Nutzerinteraktion wie möglich erfordern. Im Falle automatischer Starts und Stopps sogar gar keine, damit der Dienst nach einem ungeplanten Neustart wieder zur Verfügung steht. Das gleiche gilt für den *Default-Build-Agent*, der unter Linux standardmäßig mitinstalliert ist. Leider ist der Start von TeamCity-Komponenten etwas komplizierter als ein simpler Befehlsaufruf. So setzen diese Komponenten die Einrichtung bestimmter Umgebungsvariablen voraus, damit sie das korrekte Java-Development-Kit nutzen oder auf IPv4- anstatt auf IPv6-Adressen lauschen, wenn beide IP-Stacks im System vorhanden sind. Aus diesem Grund ist es sinnvoll, die Installation des Startskripts vorzuziehen und erst danach auf die eigentliche Installation von TeamCity einzugehen. Steht das Startskript frühzeitig zur Verfügung, geht die erstmalige Einrichtung von TeamCity deutlich leichter von der Hand.

Vor der Implementierung dieses Skripts benötigt TeamCity aber noch einen User und eine Gruppe, unter deren Identität es später laufen wird. Mit Hilfe des Debian-Tools *adduser* lässt sich der Benutzer „teamcity“ samt gleichnamiger Gruppe folgendermaßen erstellen:

```
bash:~# adduser --home=/var/lib/teamcity --system teamcity --group
Adding system user ‘teamcity’ (UID 105) ...
Adding new group ‘teamcity’ (GID 107) ...
Adding new user ‘teamcity’ (UID 105) with group ‘teamcity’ ...
Creating home directory ‘/var/lib/teamcity’ ...
```

Das Schlüsselwort *--system* stellt sicher, dass der Benutzer eine UID und GID kleiner als 1000 erhält, um ihn sicher von interaktiven Benutzern zu unterscheiden. Zusätzlich verhindert diese Option ein interaktives Login dieses Benutzers. Das Home- bzw. Arbeitsverzeichnis hat den Pfad */var/lib/teamcity*. Hier wird TeamCity später sein Datenverzeichnis *.BuildServer* anlegen, welches die Dokumentation auch als *TeamCity Data Directory* bezeichnet.

Der neue Benutzer *teamcity* erlaubt allerdings noch keinen Zugriff auf ein über SSH angebundenes Git-Repository. Leider unterstützt TeamCity nur eine Authentifizierung mittels SSH-Key-File, dass seinerseits kein Passwort haben darf. Die Weboberfläche bietet zwar weitere Authentifizierungsmethoden an, diese funktionieren jedoch alle nicht. TeamCity informiert über diesen Umstand erst bei Nutzung dieser Methoden, und das tief vergraben in den Server-Log-Dateien. Eine Möglichkeit dies zu umgehen ist ein passwortloses Key-File zu erzeugen und es bei einem geeigneten Benutzer auf dem Server mit dem Upstream-Repository zu hinterlegen. Die folgenden Befehle (mit gekürzter Shell-Ausgabe) bewerk-

stelligen das, hier exemplarisch für den Repository-User *mustermann* auf dem Host *repo*. Die Passphrase beim Aufruf von *ssh-keygen* muss **explizit leer sein**. Bei *ssh-copy-id* ist die Angabe des Passworts von Benutzer *mustermann* des Rechners *repo* allerdings notwendig.

```
bash:~# install -o teamcity -g teamcity -m 700 -d /var/lib/teamcity/.ssh
bash:~# ssh-keygen -f /var/lib/teamcity/.ssh/id_rsa
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /var/lib/teamcity/.ssh/id_rsa.
Your public key has been saved in /var/lib/teamcity/.ssh/id_rsa.pub.
...
bash:~# chown teamcity:teamcity -R /var/lib/teamcity/.ssh
bash:~# ssh-copy-id -i /var/lib/teamcity/.ssh/id_rsa.pub mustermann@repo
The authenticity of host 'repo (1.2.3.4)' can't be established.
RSA key fingerprint is 00:11:22:33:44:55:66:77:88:99:aa:bb:cc:dd:ee:ff.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'repo,1.2.3.4' (RSA) to the list of known hosts.
Password:
Now try logging into the machine, ...
```

Wie bereits angedeutet, lässt sich das Laufzeitverhalten von TeamCity über verschiedene Umgebungsvariablen beeinflussen. Die Variable `JAVA_HOME` verweist naheliegenderweise auf das zu verwendende JDK. Die `CATALINA_OPTS`-Variable enthält Kommandozeilenoptionen für die Java-VM, in welcher TeamCitys zugrundeliegender Tomcat-Server läuft. Hier, ebenso wie in den Variablen `TEAMCITY_SERVER_OPTS` und `TEAMCITY_AGENT_OPTS`, sind Optionen zur Verwendung des IPv4-Stacks wichtig. Der Build-Agent benötigt noch einen Hinweis auf den verwendeten Zeichensatz, den er sonst auf 7 Bit ASCII festlegen würde, womit das Übersetzen von Java-Applikationen fehlschlägt, sobald Umlaute im Quelltext auftauchen. Der Vollständigkeit halber sei hier eine Beispielkonfiguration angeführt:

```
JAVA_HOME="/opt/jdk1.7.0"
CATALINA_OPTS="-Djava.net.preferIPv4Stack=true"
CATALINA_PID="/opt/TeamCity/logs/teamcity-server.pid"
TEAMCITY_SERVER_OPTS="-Djava.net.preferIPv4Stack=true"
TEAMCITY_AGENT_OPTS="-Dfile.encoding=ISO-8859-15 \
-Djava.net.preferIPv4Stack=true"
```

Das nachfolgend vorgestellte Startskript (Listing 4.1) verwendet diese Variablen ebenfalls. Es ist im Stil eines System-V-Initskripts geschrieben, wie es unter Debian 6.0 üblich ist. Es reagiert dementsprechend auf die Parameter `start`, `stop` und `restart`. Es enthält jeweils einen Block von `DAEMON_`- und `AGENT_`-Variablen, die getrennt für CI-Server und Build-Agent Dateipfade, Benutzer, Gruppen und Kommandozeilenparameter festlegen. Den eigentlichen Start von TeamCity (und des Build-Agents) delegiert das Skript an das Debian-Tool *start-stop-daemon*, welches Dienste mit passenden Benutzerrechten im Hintergrund startet. Das Skript selbst hat den Pfad */etc/init.d/teamcity*, mit Besitzer und Gruppe *root* und den Unix-Rechten „744“ (für den Besitzer les-, schreib- und ausführbar, für alle anderen nur lesbar). Damit Debians System-V-Init das Skript künftig beim Hochfahren berücksichtigt, ist (einmalig) als *root* der Befehlsaufruf `insserv teamcity` notwendig.

Listing 4.1: System-V-Init-Skript für TeamCity

```
#! /bin/sh
### BEGIN INIT INFO
# Provides:          teamcity
# Required-Start:    $syslog $time $remote_fs $network mysql
# Required-Stop:     $syslog $time $remote_fs $network mysql
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: Team Integration Server
# Description:       Team Integration Server "TeamCity" from JetBrains
#
### END INIT INFO
#
# Author:    Christian Kroll <christian.kroll@tu-dortmund.de>
#

PATH="/opt/jdk1.7.0/bin:/bin:/usr/bin:/sbin:/usr/sbin"

DAEMON="/opt/TeamCity/bin/teamcity-server.sh"
DAEMON_PIDFILE="/opt/TeamCity/logs/teamcity-server.pid"
DAEMON_STARTARGS="start"
DAEMON_STOPARGS="stop"
DAEMON_USER="teamcity"
DAEMON_GROUP="teamcity"

AGENT="/opt/TeamCity/buildAgent/bin/agent.sh"
AGENT_PIDFILE="/opt/TeamCity/buildAgent/logs/buildAgent.pid"
AGENT_STARTARGS="start"
AGENT_STOPARGS="stop"
AGENT_USER="teamcity"
AGENT_GROUP="teamcity"

# Java specific settings (use upstream JDK 1.7.0, don't bind sockets to IPv6)
export JAVA_HOME="/opt/jdk1.7.0"
export CATALINA_OPTS="-Djava.net.preferIPv4Stack=true"
export CATALINA_PID=$DAEMON_PIDFILE
export TEAMCITY_SERVER_OPTS="-Djava.net.preferIPv4Stack=true"
export TEAMCITY_AGENT_OPTS="-Dfile.encoding=ISO-8859-15 -Djava.net.preferIPv4Stack=true -server"

test -x $DAEMON || exit 0

. /lib/lsb/init-functions

case "$1" in
  start)
    # starting TeamCity server
    start-stop-daemon --start --oknodo \
                      --user $DAEMON_USER \
                      --pidfile $DAEMON_PIDFILE \
                      --chuid $DAEMON_USER \
                      --group $DAEMON_GROUP \
                      --startas $DAEMON -- $DAEMON_STARTARGS

    # starting build agent
    start-stop-daemon --start --oknodo \
                      --user $AGENT_USER \
                      --pidfile $AGENT_PIDFILE \
                      --chuid $AGENT_USER \
                      --group $AGENT_GROUP \
                      --startas $AGENT -- $AGENT_STARTARGS
    ;;
  stop)
    #stopping build agent
    $AGENT $AGENT_STOPARGS
    sleep 10
    start-stop-daemon --stop --quiet \
                      --pidfile $AGENT_PIDFILE \
                      --exec $JAVA_HOME/bin/java
    rm -f $AGENT_PIDFILE

    # stopping TeamCity server
    $DAEMON $DAEMON_STOPARGS
    sleep 10
    start-stop-daemon --stop --quiet \
                      --pidfile $DAEMON_PIDFILE \
                      --exec $JAVA_HOME/bin/java
    rm -f $DAEMON_PIDFILE
    ;;
  force-reload|restart)
    $0 stop
    $0 start
    ;;
  status)
    ;;
  *)
    echo "Usage: $0 {start|stop|restart|force-reload|status}"
    exit 1
    ;;
esac

exit 0
```

4.2.2 Einrichtung des TeamCity-Servers

Der erste Schritt zur Installation des eigentlichen TeamCity-Servers beginnt mit dem Entpacken eines Tar-Archivs [14], das der Hersteller JetBrains auf seiner Webseite bereitstellt. In der Projektgruppe findet die Version 7.1.5 Verwendung. Die Installation verläuft prinzipiell ähnlich wie das simple Entpacken eines Archivs, allerdings sind ein paar zusätzliche Schritte notwendig, um korrekte Dateiberechtigungen und sowie funktionierende symbolische Links zu verschiedenen Versionen zu gewährleisten.

Das besagte Tar-Archiv entpackt unabhängig von der TeamCity-Version ein Unterverzeichnis namens *TeamCity* in den Zielstandort, weshalb es nicht sinnvoll ist direkt nach */opt* zu entpacken, da so eine bestehende Installation überschrieben würde. Sinnvoller ist es, den Inhalt des Archivs nach */tmp* zu entpacken und anschließend das so entstehende Verzeichnis */tmp/TeamCity* nach */opt/TeamCity-x.y.z* zu verschieben (hier */opt/TeamCity-7.1.5*). Der zusätzlich anzulegende symbolische Link */opt/TeamCity*, den auch das vorangegangene Startskript referenziert, zeigt schließlich auf die gewünschte Version. Zusätzlich ist nötig, rekursiv Besitzer und Gruppe des Installationsverzeichnisses auf *teamcity* zu setzen. Folgende Befehle zeigen die korrekte Installation von TeamCity in der Version 7.1.5:

```
bash:~# tar xzf /pfad/zu/TeamCity-7.1.5.tar.gz --no-same-owner -C /tmp
bash:~# chown teamcity:teamcity -R /tmp/TeamCity
bash:~# mv /tmp/TeamCity /opt/TeamCity-7.1.5
bash:~# ln -sf /opt/TeamCity-7.1.5 /opt/TeamCity
```

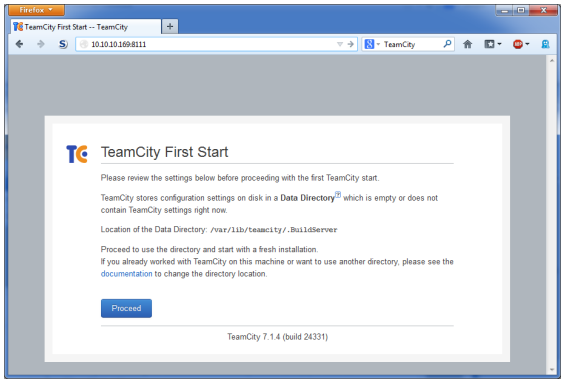
Ist TeamCity korrekt entpackt und verlinkt, geht es an die erstmalige Konfiguration. Diese läuft in zwei Stufen ab, da das bereits erwähnte Datenverzeichnis und seine Inhalte noch fehlen. Ein paar Handgriffe an der Weboberfläche von TeamCity bringen den CI-Server dazu, besagtes Verzeichnis zu erstellen und mit Leben zu füllen. Der Start des CI-Servers (als *root*) ist mit dem Startskript einfach zu bewerkstelligen:

```
bash:~# /etc/init.d/teamcity start
```

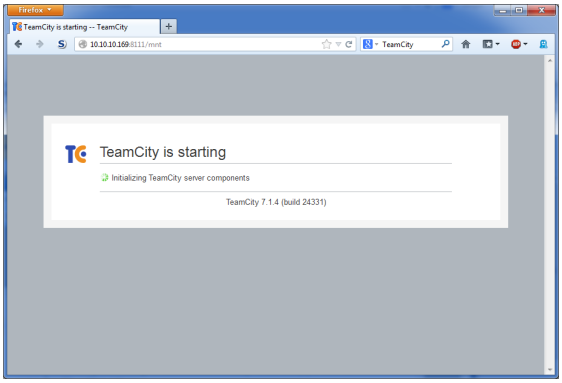
Um die Einrichtung anzustoßen, ist ein Zugriff auf die Weboberläche des Servers über TCP-Port *8111* nötig, also auf die URL *http://hostname:8111*. Es ist wichtig, dass im Browser Unterstützung für JavaScript eingeschaltet ist. Haben die bisherigen Schritte funktioniert, stellt sich ein Bild wie in Abbildung 4.1a dar. Ein Druck auf die Schaltfläche *Proceed* stößt die Initialisierung an (Abb. 4.1b), die unter Umständen mehrere Minuten in Anspruch nimmt. Dem schließt sich eine Seite mit Lizenzbedingungen an, deren Zustimmung für die weitere Installation notwendig ist. Das Häkchen für die Übermittlung anonymer Nutzerstatistiken ist hingegen nicht verpflichtend (Abb. 4.1c). Nach einem Druck auf *Continue* fragt TeamCity nach der Einrichtung eines Administrator-Benutzers (Abb. 4.1d). An dieser Stelle ist dies noch nicht sinnvoll, stattdessen ist ein Stopp des Server über die Shell nötig:

```
bash:~# /etc/init.d/teamcity stop
```

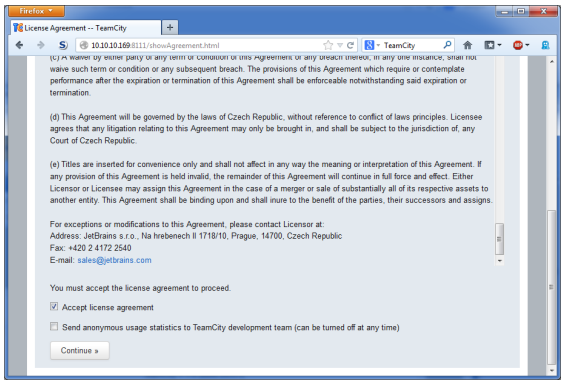
Dies ist darin begründet, dass TeamCity noch eine Verbindung mit der MySQL-Datenbank benötigt. Nach einer frischen Installation ist noch die interne HSQLDB-Lösung aktiv, eine in TeamCity integrierte, serverlose Datenbank-Engine, die laut Hersteller lediglich für Evaluationszwecke gedacht ist. Der vorangegangene, kurze Start von TeamCity ist



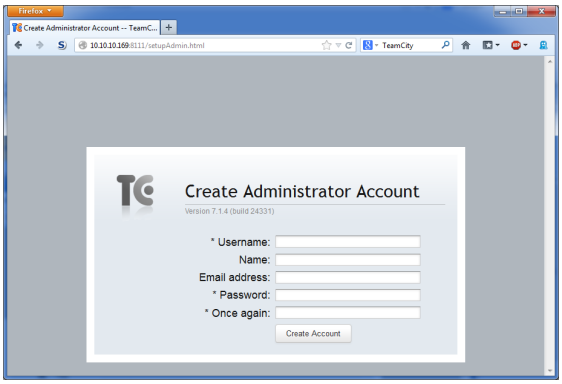
(a) Der erste Start



(b) Initialisierung



(c) Lizenzbedingungen



(d) Admin-User erstellen

Abbildung 4.1: Einrichtung von TeamCity

trotzdem notwendig, damit es das Datenverzeichnis */var/lib/teamcity/.BuildServer* anlegt und entsprechend befüllt. Von besonderem Interesse ist das nun vorhandene Verzeichnis */var/lib/teamcity/.BuildServer/config*. Dort befindet sich eine Textdatei namens *database.mysql.properties.dist*, die die Vorlage für die zu erstellende Datei *database.properties* darstellt. Am einfachsten ist es, zunächst eine passend benannte Kopie mit entsprechenden Benutzerrechten (Benutzer und Gruppe *teamcity*) zu erstellen:

```
bash:~# cd /var/lib/teamcity/.BuildServer/config/
bash:~# cp -a database.mysql.properties.dist database.properties
```

database.properties ist eine Textdatei und enthält Variablen, die folgende Werte aufweisen müssen (gegebenenfalls die entsprechenden Zeilen „unkommentieren“):

```
connectionUrl=jdbc:mysql://localhost:3306/teamcity
connectionProperties.user=teamcity
connectionProperties.password=TeamCity-Passwort
connectionProperties.useUnicode=true
connectionProperties.characterEncoding=UTF-8
```

Anstatt *TeamCity-Passwort* ist natürlich das Passwort einzusetzen, das während der Einrichtung des Datenbankbenutzers *teamcity* vergeben wurde. Nach dem Speichern der Datei benötigt TeamCity nun den Pfad zum JDBC-Treiber für MySQL. Dazu ist der Treiber im

Verzeichnis */var/lib/teamcity/.BuildServer/lib/jdbc* zu verlinken. Er befindet sich bei Debian 6.0 in */usr/share/java/mysql.jar* und der Link lässt sich wie folgt erstellen:

```
bash:~# ln -sf /usr/share/java/mysql.jar \
/var/lib/teamcity/.BuildServer/lib/jdbc
```

Für den weiteren Verlauf der Installation ist ein erneuter Start des CI-Servers über das Initskript notwendig:

```
bash:~# /etc/init.d/teamcity start
```

Auf der Weboberfläche erscheint eine Fehlermeldung, dass die Datenbank noch leer ist. Nach einem Klick auf *„I'm a server administrator, show me the details"* (Abb. 4.2a) verlangt der Server nach einem Token (Abb. 4.2b), einer mehrstelligen Nummer, die sich auf der Shell folgendermaßen ermitteln lässt:

```
bash:~# grep token: /opt/TeamCity/logs/teamcity-server.log | tail -1
INFO -  jetbrains.buildServer.STARTUP - Administrator login is required from
web UI using authentication token: 6708344064056344930
```

In diesem Beispiel ist die Nummer *6708344064056344930* in das Feld einzutragen. Ein anschließender Klick auf *Confirm* führt zu einer genaueren Beschreibung des Problems (Abb. 4.2c) und die Oberfläche bietet die Möglichkeit, die leere Datenbank mit dem Button *Proceed* zu initialisieren. Nach einer mehrminütigen Pause erscheint wieder die Lizenzseite, die wie gehabt nach einer Bestätigung verlangt (siehe die vorangegangene Abb. 4.1c), worauf ebenfalls wieder die Seite zum Anlegen eines Administrators folgt (Abb. 4.1d).

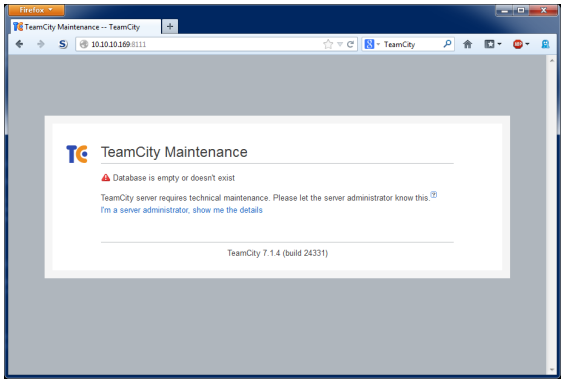
Dieses Mal ist die Einrichtung des Nutzer tatsächlich sinnvoll (Abb. 4.3a), und nach dem Anlegen schlägt TeamCity die nächsten Arbeitsschritte vor (Abb. 4.3b). Die Wahl fällt auf den Link *„configure email and Jabber settings"*, der zu den SMTP-Einstellungen führt (Abb. 4.3c). Hier ist lediglich `localhost` in das Feld *SMTP host* einzutragen mit anschließendem Klick auf *Save*. Damit ist die grundlegende Einrichtung abgeschlossen. Der Link *Projects* (in der linken Spalte) ermöglicht es, ein erstes Projekt anzulegen (Abb. 4.3d).

4.3 Bedienung von TeamCity

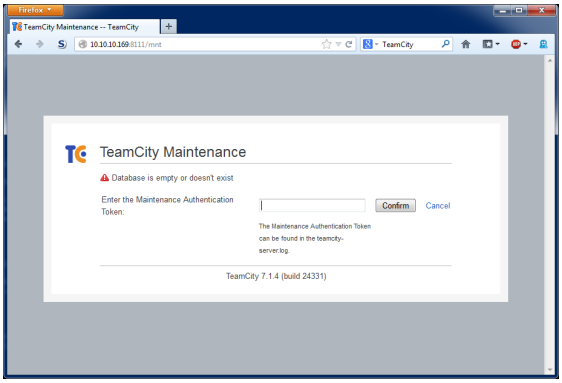
Die folgende Beschreibung kann und soll nicht mit TeamCitys offizieller Dokumentation [15] konkurrieren, zumal die Projektgruppe nur einen kleinen Teil der Funktionalität nutzt. Die eigentliche Bedienung ist zudem intuitiver als die in Abschnitt 4.2 dokumentierte Installation. Sinnvoller ist es daher, die grundlegenden Konzepte und Workflows vorzustellen, die die Handhabung von TeamCity ausmachen.

4.3.1 CI-Server und Build Agents

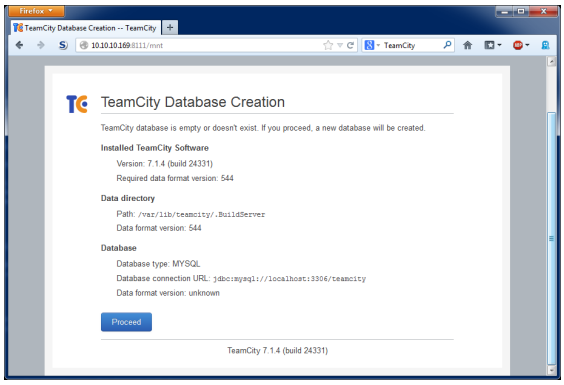
Wie bereits in den vorangegangenen Abschnitten angedeutet, umfasst eine TeamCity-Installation den eigentlichen CI-Server und mindestens einen Build Agent. Der Server verwaltet hierbei alle Metadaten, die im Zusammenhang mit den betreuten Projekten stehen.



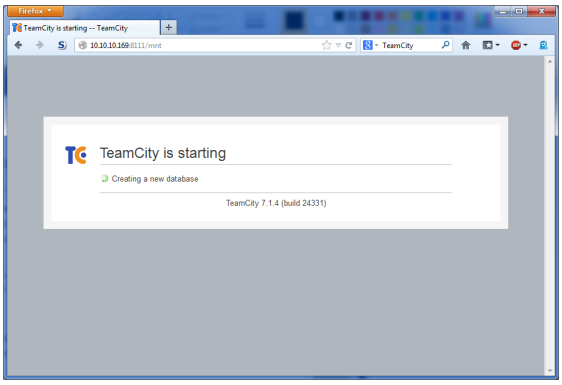
(a) Die Datenbank ist noch leer



(b) Ein Token wird verlangt



(c) Der Fehler im Detail



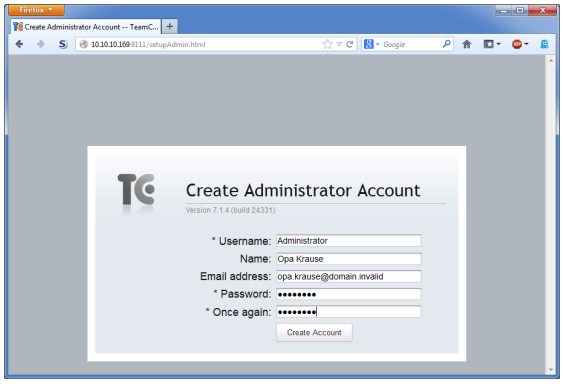
(d) Das Initialisieren der Datenbank

Abbildung 4.2: Initialisieren der Datenbank

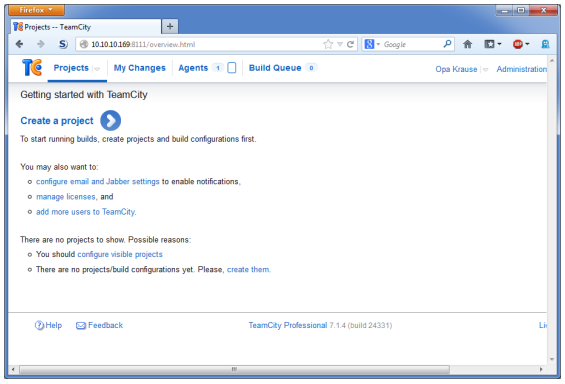
Diese Metadaten enthalten unter anderem die Adressen zu den Repositorys, exakt festgelegte Übersetzungsschritte oder die komplette Projektgeschichte in Form von Log-Dateien aller bisherigen Builds.

Build Agents stellen hingegen das ausführende Organ TeamCitys dar. Übers Netz mit dem CI-Server verbunden, setzen sie dessen projektbezogene Vorgaben zur Übersetzung und zu eventuellen Tests um. Im Anschluss übermitteln sie die Ergebnisse und Erzeugnisse eines Builds an den CI-Server. Es ist durchaus möglich, mehrere Build Agents auf verschiedene Hosts mit unterschiedlichen Betriebssystemen zu verteilen. Dies erlaubt zum Beispiel die Bereitstellung eines fertig kompilierten Projekts für mehrere Plattformen. Ferner ermöglichen mehrere Agenten eine Lastverteilung über alle von ihnen genutzten Hosts.

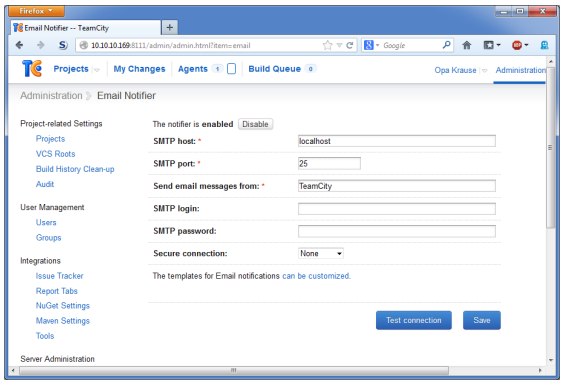
TeamCity gestattet in seiner kostenlosen „Professional“-Lizenz die gleichzeitige Nutzung von drei Build Agents an einem Server. Die Projektgruppe nutzt lediglich einen einzigen Build Agent, der auf derselben virtuellen Maschine wie der CI-Server läuft (der sogenannte *Default-Build-Agent*).



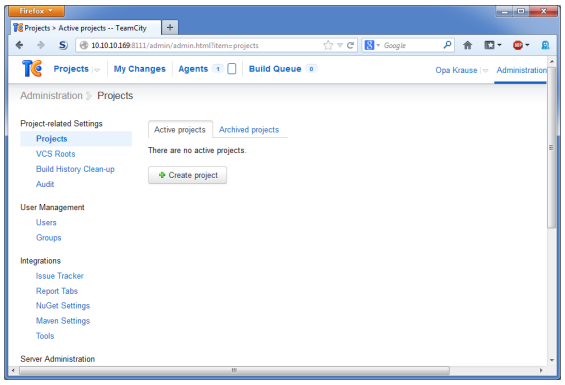
(a) Administrator-Benutzer einrichten



(b) Einstieg in Teamcity



(c) SMTP-Server einstellen



(d) Das Initialisieren der Datenbank

Abbildung 4.3: Erste Schritte

4.3.2 Abstrakte Konzepte

TeamCity bedient sich verschiedener Konzepte, um die mit Softwareprojekten assoziierten Daten zu gliedern. So fasst es Informationen dem Verwendungszweck entsprechend zu einem Datensatz zusammen, dessen Struktur einem solchen Konzept entspricht. Eindeutige Namen erlauben jeweils die Referenzierung auf diese Datensätze. Hier ist eine Auswahl dieser Konzepte:

Project In TeamCity fasst ein *Project* alle Informationen zusammen, die für Übersetzung, Test und Deploy eines realen Softwareprojekts notwendig sind. Die dort enthaltenen Informationen müssen insoweit vollständig sein, als dass die vorgenannten Schritte vollautomatisch, sprich ohne manuelle Eingriffe ablaufen können. Ein *Project* fungiert quasi als Wurzelelement, das weitere mit ihm assoziierte Datensätze anderer Konzepte referenziert.

VCS Root Ein *VCS Root* beinhaltet Informationen über Typ, Adresse, Zugangdaten und den gewünschten Branch eines Versionsverwaltungssystems. Sowohl projektbezogene als auch eine globale Deklarationen sind möglich, wobei auch hier ein eindeutiger Name als Referenz dient.

Build Runner Einzelne Build-Schritte lassen sich mit Hilfe von *Build Runnern* präzise formulieren und ausführen. Sie kapseln einerseits die Funktionalität von Build-Werkzeugen wie *ant* oder *MSBuild*, andererseits können sie aber auch mit Test-Frameworks umgehen. Wer es ganz flexibel haben möchte, kann eigene Shell-Skripte oder konkrete Anwendungen (mit Parameterübergabe) ausführen. Dies ist unter anderem für automatische Deploys nützlich. Zusätzliche Build Runner lassen sich über Plugins nachrüsten.

Build Trigger TeamCity veranlasst den Build eines Projekts erst dann, wenn mindestens ein vorher vereinbartes Ereignis eintritt. *Build Trigger* beschreiben diese Ereignisse. Der *VCS Trigger* beispielsweise veranlasst einen Build bei Änderungen im Repository. Ein *Schedule Trigger* hingegen startet einen Build zu einer fest vorgegebenen Uhrzeit.

Artifacts *Artifacts* sind die Erzeugnisse eines erfolgreichen Builds. Damit ein Build Agent diese Erzeugnisse finden kann, muss er deren Dateipfade wissen, die in seinem Arbeitsverzeichnis zu ihnen führen. Anschließend reicht er die Dateien an den den CI-Server weiter, der sie wiederum in seiner Weboberfläche für alle Projektbeteiligten zum Download anbietet. Es ist auch möglich, die Dateien während der Übertragung umzubenennen und in einer eigenen Verzeichnisstruktur zu hinterlegen.

Build Configurations Die *Build Configurations* sind die Dreh- und Angelpunkte eines *Projects*. Sie kombinieren Datensätze der vorgenannten Konzepte in der Form, dass eine konkrete, komplett durchkonfigurierte Übersetzung sowie genau definierte Tests möglich sind. Ein fest referenziertes VCS Root legt den Ort der Quellen fest und eine Liste von sogenannten *Build Steps* bestimmt Umfang und Reihenfolge der aufgerufenen *Build Runner*. Natürlich sind auch *Build Trigger* und *Artifacts* fester Bestandteil dieses Konzepts, neben vielen anderen Konzepten, auf die hier nicht weiter behandelt werden. Ein Projekt besteht aus mindestens einer Build-Konfiguration. Die kostenlose Lizenz von TeamCity erlaubt lediglich 20 *Build Configurations*, unabhängig davon, auf wie viele Projekte sie sich verteilen.

4.3.3 TeamCity-Projekt für das B-Human-Framework

Dieser Abschnitt beschreibt eine Konfiguration von TeamCity, die die Entwicklungsarbeiten am Framework überwacht. Dies behandelt ausschließlich die Linux-Variante, da bereits ein passender Build Agent durch die vorangegangene Installation eingerichtet ist. Die virtuelle Maschine, die sowohl ihn als auch den CI-Server beheimatet, muss daher alle Build-Abhängigkeiten des Frameworks erfüllen (nachzulesen in [9]), damit die Übersetzung gelingt.

Ein neues Projekt Ein Klick auf *Administration* links oben im Browserfenster führt zu der Seite in Abb. 4.4a, deren Schaltfläche *Create Project* ein neues Projekt anlegt. Daraufhin fragt die Weboberfläche nach einem Namen und einer Beschreibung für das Projekt (Abb. 4.4b). Mögliche Werte sind:

Feldname	Eingabe
Name	Framework
Description	A framework on a robot far far away...

Ein Klick auf *Create* erzeugt ein leeres Projekt und führt zu einer Übersichtsseite bezüglich noch nicht vorhandener Build-Konfigurationen (Abb. 4.4c). Der dort zu findende Link *add a build configuration* ruft eine Eingabemaske für eine neue Build-Konfiguration auf (Abb. 4.4d). Auch diese verlangt nach einem Namen und einer Beschreibung sowie nach Pfaden zu den erwarteten Build-Artefakten:

Feldname	Eingabe
Name	Linux Develop
Description	Linux-Build mit Develop-Konfiguration
Artifacts Path	Build/SimRobotHelp/Linux/Develop/libSimRobotHelp.so Build/SimRobotEditor/Linux/Develop/libSimRobotEditor.so Build/SimulatedNao/Linux/Develop/libRoboCup.so Build/qtpropertybrowser/Linux/Develop/libqtpropertybrowser.a Build/URC/Linux/Develop/URC Build/Nao/Linux/Develop/bhuman Build/bush/Linux/Develop/bush Build/ctHash/Linux/Develop/ctHash Build/libbhuman/Linux/Develop/libbhuman.so Build/cmEdit/Linux/Develop/cmEdit Build/Controller/Linux/Develop/libController.a Build/SimRobot/Linux/Develop/libSimRobotEditor.so Build/SimRobot/Linux/Develop/libSimRobotHelp.so Build/SimRobot/Linux/Develop/libRoboCup.so Build/SimRobot/Linux/Develop/libSimRobotCore2.so Build/SimRobot/Linux/Develop/SimRobot Build/SimRobotCore2/Linux/Develop/libSimRobotCore2.so

Die Schaltfläche *VCS Settings* leitet auf die Seite, die ein VCS-Root für die Build-Konfiguration festlegt (Abb. 4.4e). Ein solches existiert noch nicht, aber die Schaltfläche *Create and attach new VCS root* bietet die Möglichkeit, ein neues zu erstellen, woraufhin eine zunächst weitgehend leere Eingabemaske erscheint. Nach einer Änderung der Combo-Box *Type of VCS* auf den Wert *Git*, füllt sich die Seite wie in Abbildung 4.4f dargestellt. Als Referenzierungsmöglichkeit für das VCS-Root dient der Name „Nao Devils 2012“ und der Zugriff erfolgt auf den Git-Server „*repo*“ durch den Remote-Benutzer *mustermann*. Das Framework befindet sich in dem dort verwalteten Git-Repository */git/naodevils2011* innerhalb des Branchs *master2012*. Dies lässt sich durch folgende Werte ausdrücken (hier nicht erwähnte Felder sind so zu belassen, wie sie sind):

Feldname	Eingabe
Type of VCS	Git
VCS Root Name	Nao Devils 2012
Fetch Url	ssh+git://repo/git/naodevils2011
Default Branch	master2012
User Name Style	Author Name (John Smith)
Authentication Method	Default Private Key
User Name	mustermann

Ein Klick auf *Test Connection* verrät, ob alles funktioniert hat und führt im positiven Fall zu einer Ausgabe wie in Abbildung 4.4g. Die Schaltfläche *Save* schließt die VCS-Root-Einrichtung ab und fügt es zu der Build-Konfiguration hinzu. Auf der sich anschließenden Übersichtsseite (Abb. 4.4h) ist der *VCS Checkout Mode* auf *Automatically on agent...* zu setzen. Dies verhindert, dass der CI-Server selbst den Quelltext ausscheckt, nur um ihn anschließend zum lokalen Build Agent zu kopieren, welcher den Code genauso gut selbst auschecken kann.

Die Schaltfläche *Add Build Step* ermöglicht es schließlich, konkrete Build-Schritte anzugeben. Die dadurch aufgerufene Seite zeigt weitere Felder an, sobald in der Combo-Box *Runner Type* der Eintrag *Command Line* ausgewählt ist (Abb. 4.5a). Zuerst muss das Skript *generate* im Verzeichnis *Make/LinuxMake* des B-Human-Frameworks laufen, das weitere Makefiles erzeugt. Dies erreichen die hier gezeigten Einträge:

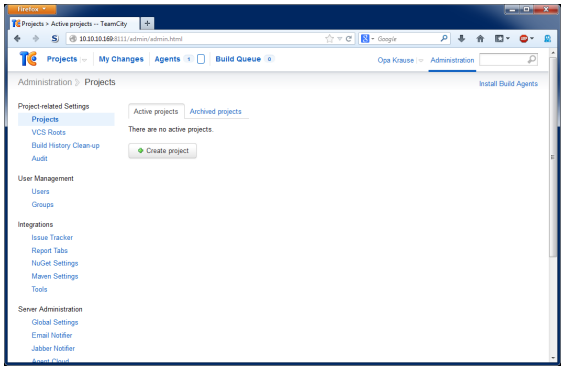
Feldname	Eingabe
Runner Type	Command Line
Step Name	Makefiles erstellen
Working Directory	Make/LinuxMake
Command Executable	/bin/bash
Command Parameters	./generate

Ein Klick auf *Save* führt zu einer Übersicht über die bisher konfigurierten Build-Schritte (Abb. 4.5b). Es fehlt allerdings noch der eigentliche Übersetzungsvorgang (der Shell-Befehl `make -j8`), der sich mit Hilfe eines weiteren Build-Schritts modellieren lässt. Die Schaltfläche *Add build Step* leitet wieder zu einer entsprechenden Seite (Abb. 4.5c), die dieses Mal folgende Einträge umfasst:

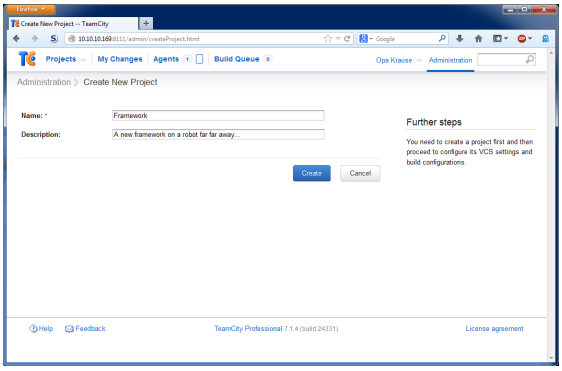
Feldname	Eingabe
Runner Type	Command Line
Step Name	Kompilieren
Working Directory	Make/LinuxMake
Command Executable	make
Command Parameters	-j8

Die Schaltfläche *Save* führt erneut zu der eben genannten Übersicht (Abb. 4.5d). Die Konfiguration der Build Trigger lässt sich mit einem Klick auf Link Nr. 5 in der rechten Spalte anstoßen. Dies führt zu einer noch leeren Übersicht über die konfigurierten Trigger (Abb. 4.5e). Die dort angebotene Schaltfläche *Add new trigger* legt einen neuen Trigger an. Im sich daraufhin öffnenden Dialog (Abb. 4.5f) ist der Eintrag in der ersten Combo-Box auf *VCS Trigger* zu setzen, sofern bei jedem Check-In eine automatische Übersetzung star-

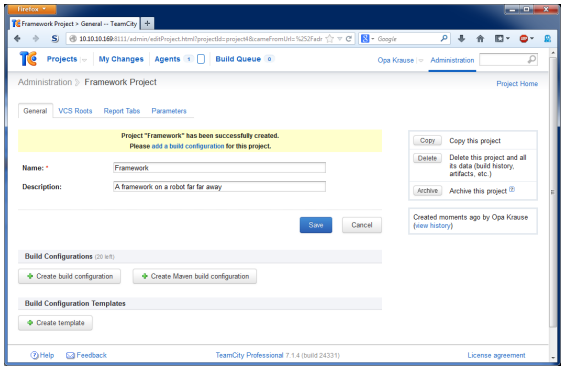
ten soll. Die Häkchen für *Trigger a build on each check-in* und *include several check-ins...* sind entsprechend einzuschalten. *Save* schließt die Einrichtung des Triggers ab, was wieder zur besagten Übersicht führt (Abb. 4.5g). Damit ist die grundlegende Konfiguration abgeschlossen. Mit der Schaltfläche *Run* oben links lässt sich der Build von Hand anstoßen. Hat es funktioniert, zeigt sich nach einer moderaten Wartezeit ein Ergebnis wie in Abbildung 4.5h. Über den *Artifacts*-Link lassen sich die Build-Erzeugnisse direkt herunterladen.



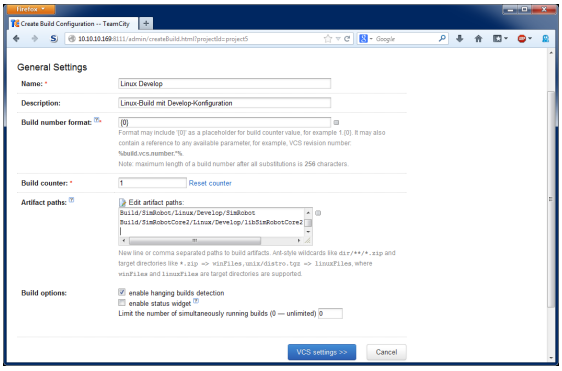
(a) Projekte administrieren



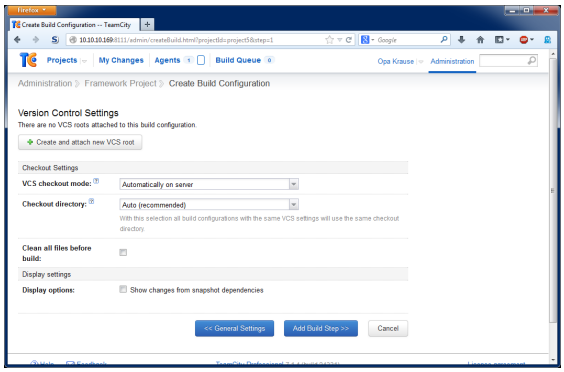
(b) Projektname und Beschreibung



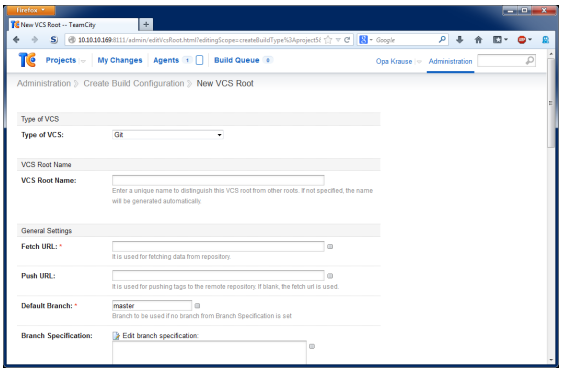
(c) Ein leeres Projekt



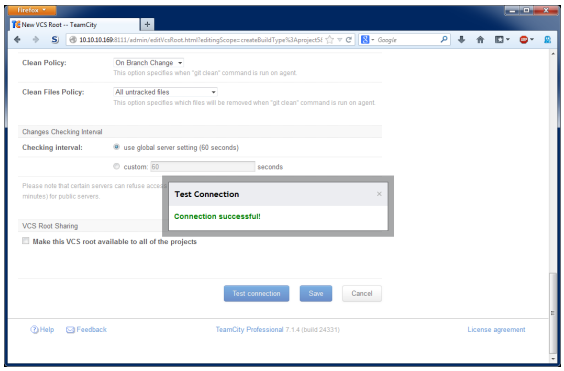
(d) Build Configuration erstellen



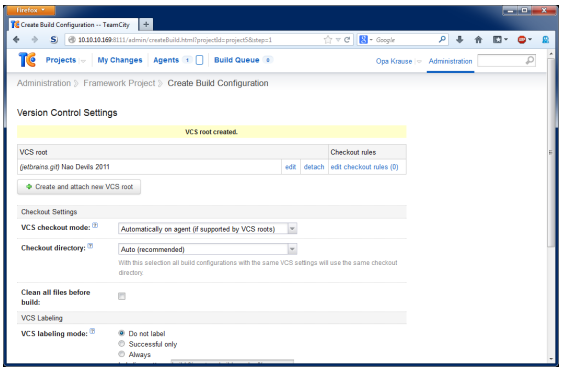
(e) VCS-Root hinzufügen



(f) VCS-Root konfigurieren

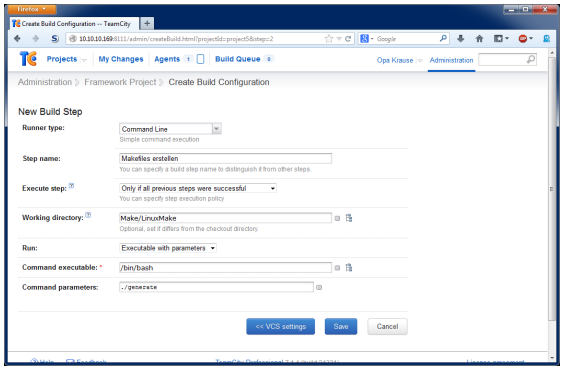


(g) Verbindung zum Git erfolgreich

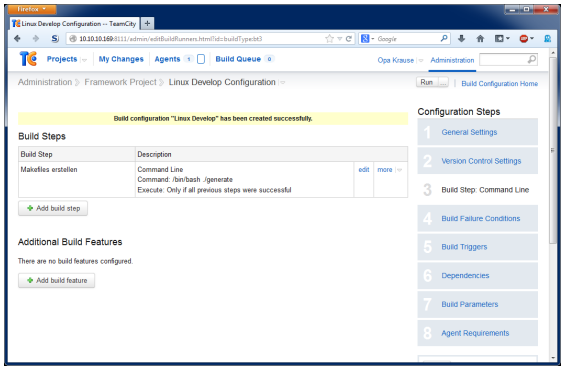


(h) VCS-Root-Einrichtung abgeschlossen

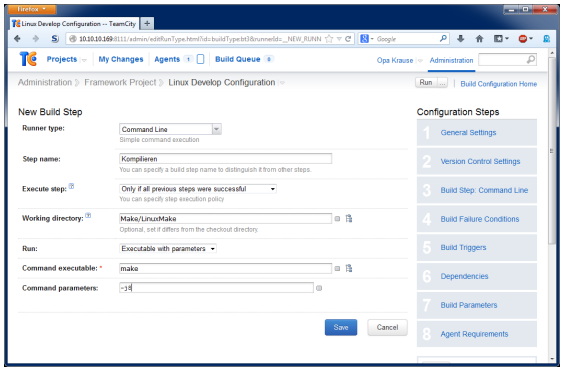
Abbildung 4.4: Ein neues Projekt



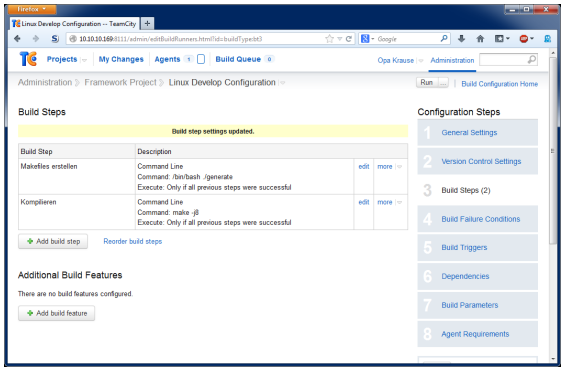
(a) Erster Build-Schritt



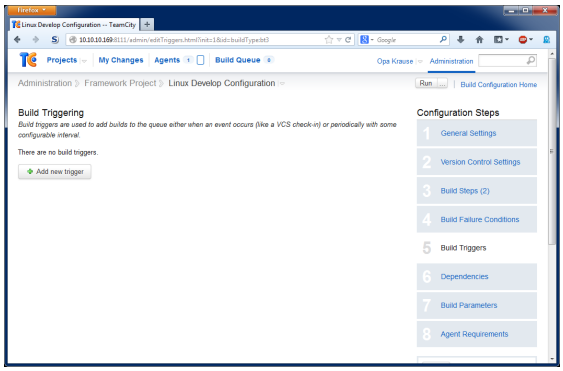
(b) Übersicht über die Build-Schritte



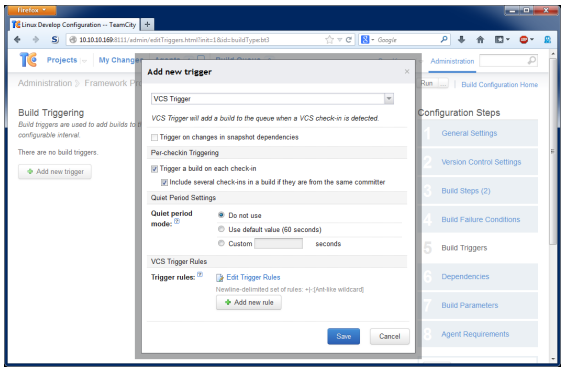
(c) Zweiter Build-Schritt



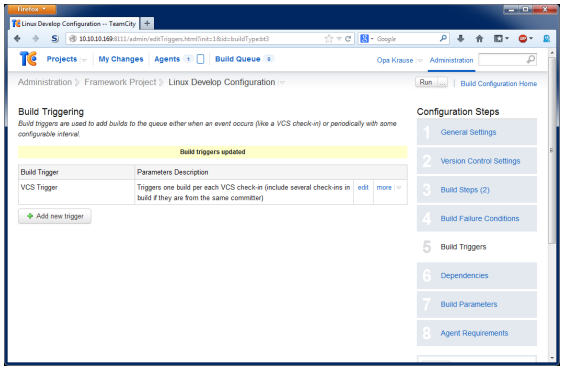
(d) Noch eine Übersicht über die Build-Schritte



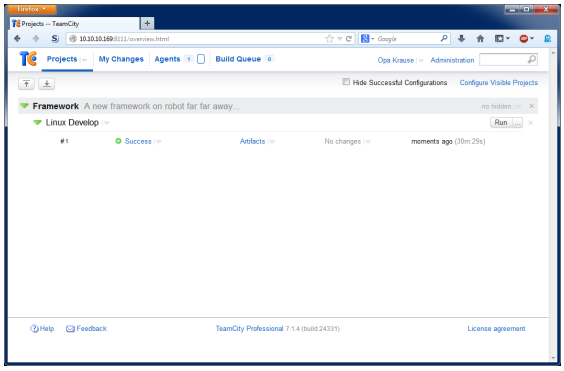
(e) Build Trigger erstellen



(f) VCS Trigger



(g) Übersicht über die Build Trigger



(h) Ein erfolgreicher Build

Abbildung 4.5: Build Steps und Trigger

Kapitel 5

Framework

bearbeitet von: Sebastian Drywa

Das neue Framework, das durch die Portierung entsteht, ist eine Weiterentwicklung des alten Frameworks, welches von den *Nao Devils* benutzt wird. Beide Framework-Versionen basieren aber auf der Implementierung von B-Human aus Bremen. Jedoch sind von den *Nao Devils* nur die Infrastruktur und die grundlegensten Funktionen des Frameworks übernommen worden. Die restlichen Module und Representations sind entweder komplett neu geschrieben oder auf die Bedürfnisse der *Nao Devils* angepasst worden.

Die Portierung des Framework und insbesondere der Module und Representations vom alten ins neue Framework ist eine aufwändige Arbeit. Viele Dateien müssen neu angepasst und andere komplett überarbeitet werden, weil sie in ihrer Funktionsweise, wie die Bremer sie benutzen, komplett anders funktionieren als sie von den *Nao Devils* gebraucht werden. Die Portierung des neuen Frameworks bringt jedoch auch viele Vorteile und nicht nur viel Arbeit mit sich. Zum einen gibt es wie oben beschrieben den neuen Simulator, welcher in Hinsicht auf die Kontinuierliche Integration gebraucht wird. Dieser ist im Gegensatz zum alten Simulator deutlich schneller geworden und kann auch Spiele mit bis zu 10 Robotern effizient simulieren. Des Weiteren bietet er auch überarbeitete und neue Funktionen. Zum Anderen bringt die Portierung auch die Vorteile einer verbesserten und aktualisierten Infrastruktur des Frameworks mit sich.

Das neue Framework ist in seiner Grundstruktur ähnlich aufgebaut wie das alte Framework. Dort sind weiterhin die Represenations und Module zu finden, die schon aus dem alten Framework bekannt sind. Die Representations kann man als Daten-Schablonen betrachten, die für ihren jeweiligen Zweck bestimmte Variablen und Methoden besitzen. So z.B. beinhalten die Representationklassen `Image.h` und `Image.cpp` die Auflösungsgrößen der Kamerabilder und Methoden, um diese Bilder von einem Farbraum in einen anderen zu konvertieren. Auch eine Definition eines Pixels im YCbCr-Farbraum ist dort zu finden. Wenn zur Laufzeit ein Objekt dieses Typs erstellt wird, können dort die Informationen, die das Framework braucht und berechnet, gespeichert werden. Die Module hingegen sind die ausführenden Elemente im Framework. Sie erhalten die Representations als Input-Daten von vorherigen Modulen, benutzen diese Daten, um ihre eigenen Aufgaben auszuführen und speichern die Ergebnisse am Schluss in eigene Representations ab oder aktualisieren die Input-Representations. Dann werden die Representations, welche neue Informationen enthalten, an die nächsten Module als Output-Daten weitergeschickt. Zu diesen Modulen

gehören z. B. das Image-Processing für die Bildauswertung, die Walking-Engine, die die Bewegungen des Roboters berechnet und die Self-Localisation, die die Postion des Roboters auf dem Feld bestimmt. Des Weiteren sind viele Strukturen und Basisfunktionen durch Bremen in der Infrastruktur des Frameworks verbessert und überarbeitet worden. Hierzu gehört z. B. die Stream-Klasse, die für das streaming der Variablen im Framework verantwortlich ist.

Die meisten Module und Representations, wie auch die oben genannten des neuen Frameworks, werden gegen Module aus dem *Nao Devils* Framework ersetzt, da entweder so viele Änderungen an den Modulen vorgenommen oder die Module komplett selbst geschrieben worden sind. Es ergibt daher keinen Sinn, diese im neuen Framework anzupassen. Darum ist es besser, schneller und auch einfacher, die ausgetauschten Module an das neue Framework anzupassen. Die anderen Module und Representation, die nicht im neuen Framework ausgetauscht werden, werden so umgeschrieben, wie sie für die Zwecke im neuen Framework gebraucht werden. So müssen z. B. die Module für die Kamerabilder umgeschrieben werden, damit beide Kameras gleichzeitig funktionieren und Bilder zur Verfügung stellen können. Dies ist eine komplett andere Methode der Kameranutzung als die von Bremen, die in ihrem Framework die Kameras nicht gleichzeitig benutzen, sondern immer zwischen den beiden Kameras nach bestimmten Kriterien hin und her wechseln. Ihr Image-Processing bekommt dadurch auch nur ein Bild pro Frame zur Verfügung gestellt, während im Dortmunder Image-Processing zwei Bilder pro Frame bearbeitet werden - von beiden Kameras jeweils ein Bild. Diese und weitere Grundarbeiten müssen jedoch erst gemacht werden, bevor die restlichen Module und Representations portiert werden können.

Bevor jedoch mit der Portierung begonnen werden konnte, musste erst einmal ein Grundverständnis für das neue Framework entwickelt werden. Dies war nicht so leicht, da die Menge an Klassen und Code einen förmlich erschlagen hat und gar nicht genau wusste, wo genau man mit der Arbeit beginnen sollte. Zum Glück jedoch gab es ein Hilfsmittel, welches benutzt werden konnte. Durch Zuhilfenahme der Dokumentation des Code-Release von 2012 [9] konnte ein relativ gutes Verständnis der Arbeitsweise des Frameworks entwickelt werden. Und so ist es möglich gewesen, nach kurzer Anlaufzeit die ersten Probleme und Anpassungen anzugehen.

In den folgenden Abschnitten werden nun kurz ein paar der Unterschiede und Anpassungen vom alten zum neuen Framework vorgestellt und erläutert, warum diese nötig sind.

5.1 Mathe-Klassen

Bei der ersten Sichtung des Codes des neuen Frameworks von Bremen fiel sehr schnell auf, das die meisten Variablen, die im aktuellen Framework noch als `double` deklariert waren, nun nur noch als `float` deklariert sind. Auch die selbst erstellten Mathe-Klassen, deren Methoden `double` als Parameter benutzen, sind nun durch `float` ersetzt worden. Da die meisten Modul-Klassen im Framework die selbst erstellten Mathe-Klassen jedoch inkludieren, ist beschlossen worden, dass an dieser Stelle mit der Portierung begonnen werden soll.

Jedoch ist schnell zu erkennen gewesen, dass dies keine leichte Aufgabe wird. Viele der Mathe-Klassen im neuen Framework sind abgeändert und überarbeitet, teilweise sogar aufgesplittet worden und alle Templates hatten nun als Standardwert den Wert `float` implementiert. Auch hat B-Human neue Klassen hinzugefügt und alte Klassen, die nicht mehr benötigt werden, aus dem neuen Framework entfernt. Die Dortmunder Module hingegen benutzen durchgängig `double`, dessen Genauigkeit an vielen Stellen für bestimmte Module auch gebraucht und vorausgesetzt wird. Wie die neuen Mathe-Klassen von Bremen sind auch die Dortmunder Mathe-Klassen an vielen Stellen abgeändert und überarbeitet worden. Es sind auch neue Methoden, die benötigt wurden, hinzugefügt worden. So ist es nicht möglich ohne großen Aufwand, die Mathe-Klassen aus beiden Frameworks in einem Framework zusammenzufassen.

Die Lösung, die nun benutzt wird, ist simpel wie auch einfach. Die Mathe-Klassen aus beiden Frameworks werden in das neue Framework integriert. So ist gewährleistet, dass die Module des neuen Frameworks weiterhin die Mathe-Klassen des neuen Frameworks benutzen können, während die Dortmunder Module, die portiert werden weiterhin die Dortmunder Mathe-Klassen benutzen können.

Zu diesem Zweck wird zuerst im alten Framework der Ordner „Math“ mit den Mathe-Klassen in „MathDortmund“ umbenannt und alle Klassennamen bekommen den Zusatz „_D“ damit sie im neuen Framework von den Mathe-Klassen von B-Human unterschieden werden können. Als nächstes werden im ganzen alten Framework die Namen und includes der Mathe-Klassen so abgeändert, dass sie zu den neuen Namenskonventionen der Mathe-Klassen passen. So wird schon einmal sichergestellt, dass alle Module aus dem alten Dortmund Framework die Dortmunder Klassen benutzen können. Danach kann der Mathe-Ordner in das neue Framework eingebunden werden.

Bei der Portierung muss durch diese Vorarbeit nur noch auf eine Besonderheit geachtet werden. Wenn ein Modul oder eine Variable, die aus dem neuen Framework stammt, in einem Dortmunder Modul benutzt wird, kann dort ein Fehler zwischen den beiden Mathe-Klassen auftreten. Dies kann passieren, wenn das eine Modul die Bremer Klassen benutzt und das Dortmunder die Dortmunder Mathe-Klassen. Zu diesem Zweck und wenn es notwendig ist, werden in den Dortmunder Mathe-Klassen dann entsprechende Cast-Operatoren und Konstruktoren geschrieben, um diese Konflikte aufzulösen.

5.2 *CABSL*

Bei der Portierung hat sich im Bereich des Verhaltens das meiste geändert. Hier wird die neue Verhaltenssprache *CABSL* integriert, welche komplett aus C-Makros aufgebaut ist und die alte Verhaltenssprache *XABSL* ersetzt, die bis jetzt im alten Framework verwendet worden ist.

Der Grund, warum man nun auf eine neue Verhaltenssprache wechselt, ist das man mit einigen Aspekten von *XABSL* Schwierigkeiten hat. Zum einen gibt es das Problem der Restriktionen, die einem einen gewissen Grad an Freiheiten beim Programmieren entziehen. Wenn man Representations, Variablen oder Methoden, welche im Framework als `c++` Code vorliegen und dort benutzt und aktualisiert werden, in *XABSL* verwenden will, muss

man diese erst in sogenannten *Symbolen* kapseln. Anders hat *XABSL* keinen Zugriff auf diese Variablen und deren Werte und es wäre nicht möglich mit ihnen in den Options und States zu arbeiten. Zum anderen braucht man für *XABSL* einen Compiler, womit das implementierte Verhalten später in den entsprechenden Code umgesetzt werden kann. Dieser Compiler wird seit Jahren genauso wie die Sprache selber nicht weiterentwickelt.

Jedoch ist die Grundidee und Grundstruktur von *XABSL*, welche auf Zustandsautomaten und situationsabhängigen Übergängen basiert, eine gute Ausgangsbasis für die Verhaltensentwicklung und erleichtert diese enorm (siehe Listing 5.1).

Listing 5.1: Beispiel für XABSL-Option „initial_state“

```
option initial_state
{
  initial state sit_down
  {
    decision
    {
      if(key.chest_button_pressed_and_released ||
         soccer.disable_pre_initial)
        goto stand_up;
      else
        stay;
    }
    action
    {
      soccer.behavior_action = penalized;
      motion.type = special_action;
      motion.special_action = play_dead;
    }
  }
}
```

Die gleichen Überlegungen hat auch Bremen gehabt und beschlossen *XABSL* durch eine selbst programmierte Sprache zu ersetzen. Diese Sprache nennen sie „State Machine Behavior Engine“ (*SMBE*) und sie baut auf den Grundkonzepten von *XABSL* auf, soll aber einige der Schwächen und Fehler davon lösen bzw. korrigieren. So braucht *SMBE* keine Symbole mehr, um mit Representations und Variablen zu kommunizieren, sondern der c++ Code kann direkt in den Options and States benutzt werden. Jedoch hat diese Sprache genauso wie *XABSL* einen Nachteil. Es wird für *SMBE* auch wieder einen Compiler benötigt, der die Sprache interpretieren und in entsprechenden Code umwandeln kann.

Auch wenn die Sprache ihre Vorteile gegenüber *XABSL* hat, ist entschieden worden, diese bei den *Nao Devils* nicht zu benutzen. Denn im Gegensatz zu *XABSL*, hinter dem noch eine Community steht, welche man bei Problemen fragen kann, gibt es bei *SMBE* nur Bremen, die diese Sprache weiterentwickeln können. Dieses Risiko will man nicht eingehen. Denn man würde nur eine Sprache, die nicht weiterentwickelt wird gegen eine andere tauschen. Bei den RoboCup GermanOpen 2013 hat sich diese Skepsis der Sprache *SMBE* gegenüber dann am Schluss auch als richtig erwiesen. Bremen hat diese neue Verhaltenssprache nach

einem Jahr schon wieder verworfen und stattdessen eine neuen Sprache entwickelt - *CABSL* .

CABSL hat die gleiche Grammatik und Syntax (siehe Listing 5.2) wie *XABSL* und *SMBE*, ist aber dennoch keine eigene Verhaltenssprache im eigentlichen Sinne mehr. Wenn man die beiden Listings vergleicht, erkennt man, das die beiden Sprachen eine ziemliche Ähnlichkeit miteinander haben.

Listing 5.2: Beispiel für CABSL-Option „Start“

```
option(Start)
{
  initial_state(playDead)
  {
    transition
    {
#ifdef TARGET_SIM
      goto standUp;    // Don't wait for the button in SimRobot
#endif

      if(action_done)  // chest button pressed and released
        goto standUp;
    }

    action
    {
      specialAction(SpecialActionRequest::playDead);
    }
  }
}
```

CABSL ist komplett in `c++` Code geschrieben und besteht nur aus C-Makros, was viele Vorteile mit sich bringt. Zum einen kann auf einen Compiler komplett verzichtet werden, da das ganze Verhalten komplett wie erwähnt in `c++` geschrieben und dadurch wie jeder andere `c++` Code auch übersetzt werden kann. Zum anderen kann man wie auch schon in *SMBE* `c++` Code direkt in die States und Options schreiben, da diese ja aus Makros bestehen. Den größten Vorteil, den *CABSL* jedoch mit sich bringt, ist die Möglichkeit, das geschriebene Verhalten wie jeden anderen Code zu debuggen. Man kann dadurch den kompletten Weg durch das Verhalten nachvollziehen und so direkt sehen, warum z.B. eine falsche Entscheidung getroffen wird oder wo man bei der Implementierung einen Fehler gemacht hat. Bei *XABSL* hat man dafür nur das Debug-Tool zur Verfügung, um solche Fehler zu erkennen und zu beheben.

Trotz der ganzen Vorteile, die *CABSL* mit sich bringt, ist dies eine neue Sprache. Daher zeigen sich auch Nachteile dieser Sprache. So gibt es für diese Sprache noch keine Tools, die helfen, das geschriebene Verhalten zu verbessern. Bei *XABSL* gibt es auch z.B. ein Tool, das aus dem implementierten Verhalten einen kompletten Abhängigkeitsgraphen der Options und States zeichnen kann. Dies wird auch für *CABSL* auf lange Sicht wünschenswert sein.

5.3 TeamComm

Einer der wichtigsten Teile bei der Portierung des Frameworks ist die Integration der TeamComm ins neue Framework. Unter der TeamComm versteht man die Kommunikation der Roboter untereinander. Im Framework kann für die TeamComm angegeben werden, welche Representations genau an die anderen Roboter gesendet werden sollen. Dies ist in Hinsicht auf das Verhalten und der Lokalisierung sehr wichtig, da mehr als 60% dieser Module auf die Daten der anderen Roboter angewiesen sind, um vernünftige Ergebnisse zu berechnen und daraus Entscheidungen zu treffen.

Die TeamComm ins neue Framework zu integrieren, ist jedoch kein großes Problem, da diese im Framework schon implementiert ist. Die für die TeamComm benötigten Dateien wie die Socket-Klassen, die für das versenden und empfangen der UDP-Pakete verantwortlich ist, und die Händler-Klassen, welche für die Interpretation der empfangenen Pakete nötig sind, sind schon im neuen Framework vorhanden und müssen auch nicht groß angepasst werden.

Die TeamComm-Implementierung von B-Human ist aber über das ganze Framework verteilt, für die Dortmunder TeamComm ist jedoch eine zentrale Stelle gewünscht. Dazu muss aus dem neuen Framework die derzeit implementierte TeamComm gegen die zentralisierte TeamComm der Dortmunder ausgetauscht werden. Hierzu werden im neuen Framework alle Module gesucht, in welchen das Makro `TEAM_OUTPUT` zu finden ist. Diese Makros können dann entweder gelöscht werden, wenn das Modul beim alten Framework nichts übertragen hat oder gar nicht mehr verwendet wird, oder sie werden in die Klasse `TeamDataProvider.cpp` verschoben, wo die zentralisierte TeamComm nun unter der Methode `handleTeamCommSending()` zu finden ist. Wie genau die TeamComm aufgebaut ist und diese funktioniert, wird später ausführlicher im Kapitel Challenges bei der DropIn-Challenge(siehe Kapitel 13.1) erläutert.

Optional ist es möglich, in der Klasse `TeamHandler.cpp` bei der Methode `receive()` durch ein paar Zielen Code eine kleine Logging-Datei anzulegen und zu speichern. Diese kann dazu verwendet werden, um Probleme bei der Übertragung der Pakete oder im WLAN ausfindig zu machen wie z. B. ob Pakete verloren gegangen oder ob diese zu spät angekommen sind. Und auch die Größe jedes einzelnen Paketes von jedem Roboter kann man betrachten. Dieses optionale WLAN-Logging ist beim RoboCup 2013 zum Einsatz gekommen, um zu überprüfen, wie gut die WLAN-Qualität in der Halle war.

5.4 Ausblick

Der letzte Schritt der Portierung wird im Laufe der Zeit die Optimierung des Frameworks sein. Zu dieser Optimierung gehört zum einen das Aufräumen des Frameworks, da noch viele Altlasten von Bremen dort enthalten sind. Zu diesen Altlasten gehört z. B. der `StateMachineCompiler`, welcher vorher von Bremen zum compilieren der State Machine Behavior Engine genutzt wurde. Auch finden sich im Code noch Representations und Module, die Bremen in ihrem Framework benutzt. Im neuen Framework haben diese jedoch keinen Zweck mehr und können entfernt werden.

Zum anderen sollen die Mathe-Klassen noch optimiert werden, da es auf Dauer immer wieder zu Problemen bei der Umwandlung von den Dortmunder zu den Bremer Klassen kommen kann oder umgekehrt. Eine Möglichkeit dafür wäre, die Dortmunder Mathe-Klassen nur noch als erbende Klassen mit den Bremer Klassen als Basis zu benutzen. Dies hätte den Vorteil, dass bei einer erneuten Portierung die Dortmunder Klassen einfacher übernommen werden können.

Dennoch ist die Portierung auf das neue Framework seit dem RoboCup auch ohne diese Optimierungen abgeschlossen und hat seinen ersten Einsatz auf dem RoboCup mit Erfolg gemeistert.

Kapitel 6

NaoDeployer

bearbeitet von: Jan-Hendrik Berlin

Bei der Teilnahme am Workshop *RoBoW 12.3* in Berlin hat sich gezeigt, dass das Aufsetzen mehrerer Roboter per Kommandozeile unübersichtlich ist. Um Fehleingaben zu vermeiden und den Umgang zu erleichtern wurde das Werkzeug „NaoDeployer“ entworfen. Der „NaoDeployer“ wurde in C# entwickelt, da das Framework der NaoDevils in Microsoft Visual Studio entwickelt wird. In der ursprünglichen Version 1 wurden nur das Aufsetzen der Roboter berücksichtigt, die Version 2 wurde weitreichend erweitert um den allgemeinen Umgang mit den Robotern zu erleichtern.

6.1 NaoDeployer Version 1

Einleitung Bisher wurden die Roboter mit dem Kommandozeilen-Skript `copyfiles.cmd` mit neuem Programmcode bespielt. Über bestimmte Parameter konnte die IP-Adresse, der Name des Roboters sowie der Verzeichnisnamen für den Robotercode eingestellt werden. Durch eine grafische Oberfläche sollte die Nutzerfreundlichkeit erhöht werden und das Aufsetzen beschleunigt werden. Bei der Analyse des bisherigen Ausetzskriptes wurden weitere Parameter für die Teamfarbe, die Spielernummer, das Location-Profil und die Möglichkeit das Verzeichnis auf dem Roboter vor dem Kopieren zu leeren entdeckt. Alle Funktionen sollten in einer einfachen Oberfläche abrufbar sein.

Architektur Das Programm besteht aus einem WindowsForms Fenster und einer Klasse „deploy“. Die GUI ist recht funktional aber auch intuitiv gehalten (siehe Abb. 6.1). Sie erfüllt die Anforderungen.

Die Klasse „deploy“ besteht aus einem Konstruktor und einer Funktion „DoDeploy“. Im Konstruktor wird die Dateien `LAN.conf` geparsed und die Verzeichnisse in „..\naodevils2011\Config\Locations“ gelistet. Die Verzeichnisse sind die Locations welche aktuell zur Verfügung stehen. Da die Roboter sowohl über LAN wie auch über WLAN zu erreichen sind, dort aber unterschiedliche IP-Adressen haben, ist es nötig dies zu unterscheiden. Die Erfahrung der *RoBoW 12.3* hat gezeigt, dass man auf Turnieren etc. andere

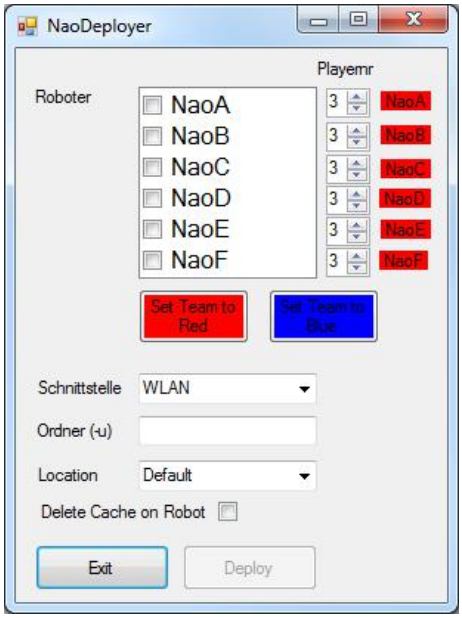


Abbildung 6.1: GUI des NaoDeployer Version 1

Adressen bekommt. Aus diesem Grund kann man beliebig viele Schnittstellen anlegen. Schnittstelle ist also als Adressbereich zu interpretieren.

Die Datei `LAN.conf` enthält die IP-Adressen und Schnittstellen der Roboter. Es ist eine einfache Textdatei, in welcher `#` ein Kommentar ist und `*` eine Schnittstelle des Roboters definiert. Unter den Schnittstellen listet man Zeile für Zeile die IP-Adressen der Roboter auf. Die Erste Adresse ist für Roboter NaoA, die zweite für NaoB ... die letzte für NaoF. Die Funktion „DoDeploy“ hat einige Parameter (siehe Listing 6.1).

Listing 6.1: Parameter der Funktion „DoDeploy“

```
public void DoDeploy(int intf,  // number of the interface
                     bool [] b, //Array of bools, if true,
                        the nao gets a new copy of software
                     string u,  // directory name on the robot
                     bool delete, // if true, delete cache on
                        robot
                     int[] player, // the player numer
                     bool [] Team, // false = blue, true =
                        red Team
                     int loc)   // number of the location
                              )
```

Im wesentlichen werden Daten übergeben, für welchen Roboter das Script `copyfiles.cmd` mit welchen Optionen aufgerufen wird. Die Funktion erstellt dann anhand der übergebenen Daten einen String zum Aufruf des Scripts und führt das Script dann aus.

Zwischenfazit Der NaoDeployer war in dieser Version gut nutzbar und erleichterte die Arbeit. Es wurden aber noch andere Features erwünscht. Es sollte möglich sein SSH-Befehle zum starten/stoppen des Frameworks zu senden. Des weiteren wäre es schön Logfiles unkompliziert vom Roboter laden zu können. Seit Februar 2013 besteht für das Team die Möglichkeit weitere Naos auszuleihen. Somit wurde es nötig die Anzahl der Roboter im Programm zu erhöhen. Da man auf den geliehenen Roboter erst mit das Framework der NaoDevils installieren muss, wäre es von Vorteil, wenn dies auch über den NaoDeployer funktionieren würde.

6.2 NaoDeployer Version 2

Änderungen der Architektur Als erstes wurden im Programm einige String Arrays durch Listen ersetzt um die variable Anzahl von Robotern zu ermöglichen. Dies wurde in der Klasse „deploy“ vorgenommen. Die geänderten Parameter der Funktion „DoDeploy“ verdeutlicht die wesentlichen Änderungen (vergl. Listing 6.2).

Listing 6.2: Parameter der überarbeiteten Funktion „DoDeploy“

```
        public void DoDeploy(   int intf, // number of the
            interface
                                List<bool> b, //List of
                                    bools, if true, the nao
                                    gets a new copy of software
                                string u,  // directory name
                                    on the robot
                                bool delete, // if true,
                                    delete cache on robot
                                List<int> player, // the
                                    player numer
                                List<bool> Team, // false =
                                    blue, true = red Team
                                int teamnumber, // teamnumber
                                int loc, // number of the
                                    location
                                string compileOpt, //
                                    compiler option
                                Queue<string> output, //
                                    queue for output log (more
                                    details then error)
                                Queue<string> error   //
                                    queue for error log
                                )
```

Die Datei `LAN.conf` wird nach wie vor geparsed. Jetzt entscheidet aber die Anzahl der IP-Adressen einer Schnittstelle, wie viele Roboter in der GUI angezeigt werden.

Die alte GUI (siehe Abb. 6.1) war nicht auf die neuen Anforderungen anzupassen. Variable NumericUpDown Elemente für die Spielernummer einzubauen ist nicht möglich. Die neue GUI (siehe Abb. 6.2) hat folgenden Aufbau. Die Fläche des Fensters wurde vergrößert und ein Spielfeld wurde an der rechten Seite hinzugefügt. Dort kann man nun an 10 Positionen durch comboBoxen Roboter aufstellen. Die Positionen symbolisieren sowohl Teamfarbe als auch Spielernummer. Einige neue Buttons wurden eingefügt um die gewünschten Funktionen zu aktivieren.

Abbildung 6.2: Hauptfenster des NaoDeployer Version 2

Des weiteren wurden neue WindowsForm Fenster dem Projekt hinzugefügt. In einem Fenster kann man den Pfad zum Ordner naodevils2011 setzen (siehe Abb. 6.3). Es ist zwingend notwendig, dass das Programm den Pfad zu diesem Ordner kennt, denn in diesem Ordner befinden sich die zu überspielenden Daten. Deswegen wird beim Start des Programms geprüft ob der Pfad korrekt ist, wenn nicht, wird dieses Config-Fenster gezeigt (siehe Abb. 6.3). Es kann erst verlassen werden, wenn der Pfad korrekt ist.

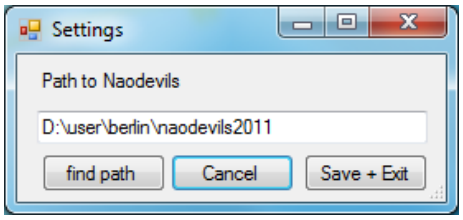


Abbildung 6.3: Config Fenster des NaoDeployer Version 2

Das Fenster zum downloaden der Logfiles listet den Inhalt des Verzeichnisses mit den Logfiles in einer checkedListBox auf (siehe Abb. 6.4). Man kann so für jeden Roboter wählen welche Logfiles man kopiert haben möchte und im letzten Schritt das Kopieren starten.

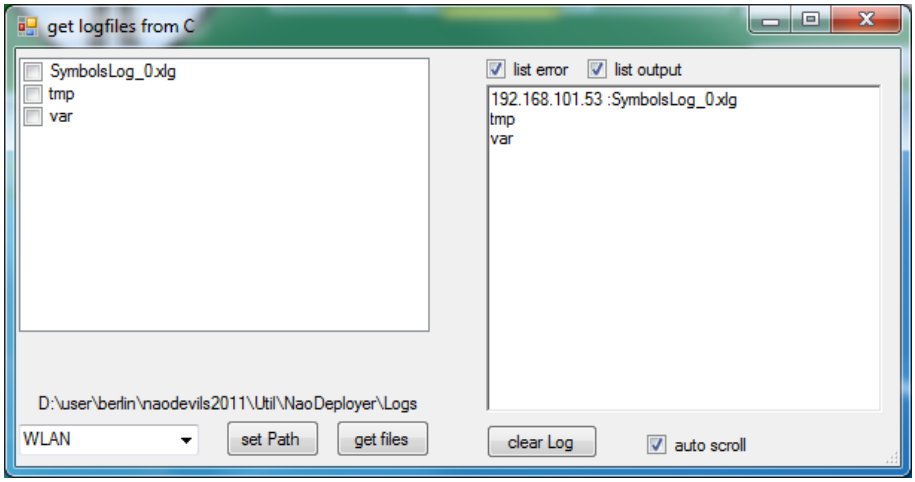


Abbildung 6.4: Download Fenster des NaoDeployer Version 2

Ein Fenster heißt „framework setup“ und ist im Hauptfenster über den Button „setup robot“ erreichbar (siehe Abb. 6.5). Es dient dazu auf einen Roboter die Basisinstallation des Frameworks zu starten.

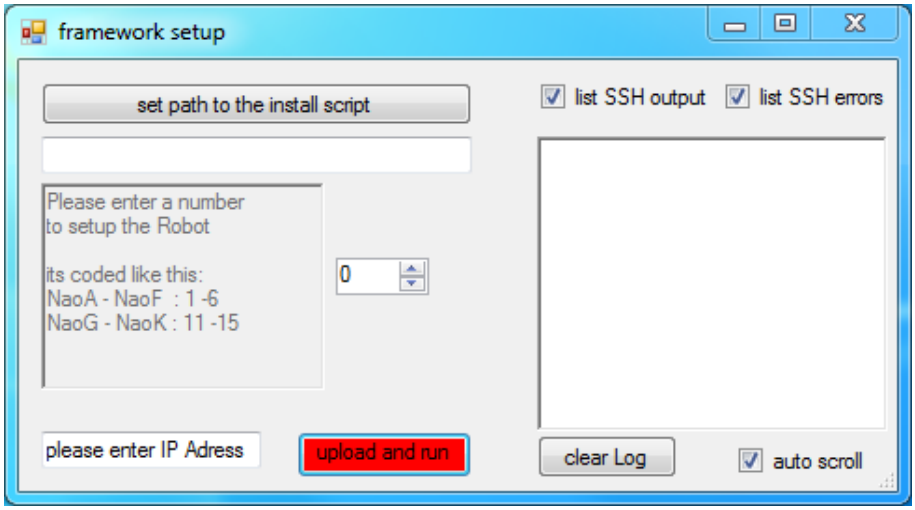


Abbildung 6.5: „setup robot“ Fenster des NaoDeployer Version 2

Über den Button „run list of commands“ öffnet man ein Fenster (siehe Abb. 6.6) mit in dem man eine Liste von bash-Befehlen eingeben kann. Die Befehle werden auf jedem zuvor ausgewählten Roboter ausgeführt.

Um dem Benutzer eine Rückmeldung zu geben, ob ein Befehl ausgeführt wurde oder ob das Script `copyfiles.cmd` mit Fehlern beendet wurde, gibt es ein weißes Textfeld auf der rechten Seite der betreffenden Fenster. Über einen Timer wird alle 500ms der Inhalt verschiedener Warteschlangen in das Textfeld verschoben. Bei dem Aufruf einer Funktion der Klasse „NDDssh“ oder der Funktion „doDeploy“ (vergl. Listing 6.2) werden die entsprechenden Warteschlangen übergeben.

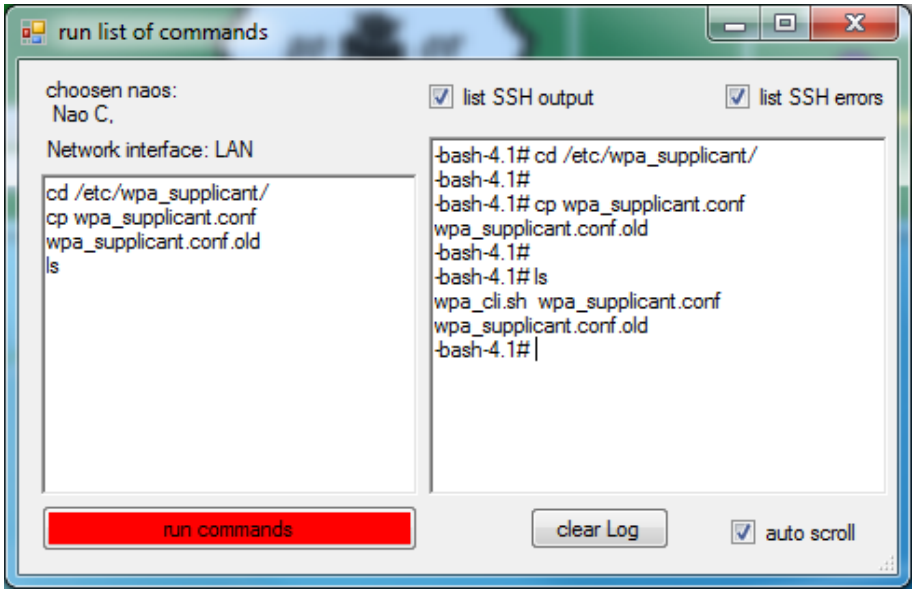


Abbildung 6.6: „run list of commands“ Fenster des NaoDeployer Version 2

Die neu angelegte Klasse „NDDssh“ ermöglicht den Zugriff via SSH auf die Roboter. Es wurde eine Bibliothek verwendet, welche ssh implementiert, statt externe Tools wie „Putty“ oder „ssh“ zu nutzen. Die Bibliothek heißt `Renci.ssh.net` [1]. Es ist sowohl möglich Befehle auf der bash auszuführen als auch sftp zu nutzen. Auf den Robotern muss man sich für teilweise als normaler Benutzer anmelden und sich dann Superuser-Rechte mit dem Befehl `su` hohlen. Die Superuser-Berechtigung kann man nur mittels „keyboardinteraktiver“ Eingabe erhalten, was heißt dass man Passwörter mit der Tastatur eingeben muss. Die Bibliothek erlaubt es einen sshStream auf zu machen, sodass die Gegenseite meint man würde „keyboardinteraktiv“ agieren. Die Ausgaben können abgefangen und verarbeitet werden. Geworfene Exeptions können wie in C# üblich gefangen werden. Die Bibliothek bietet somit bessere Möglichkeiten als ein Aufruf von „Putty“ mit Parametern.

In der Klasse „NDDssh“ gibt es Funktionen um Befehle auf einem Roboter auszuführen, ein Verzeichnis aufzulisten oder Dateien zu kopieren. Die Login Daten sind in der Klasse eingebettet, da sie für alle Roboter gleich sind.

6.3 Fazit und Ausblick

Das Programm ist umfangreicher geworden, als Anfangs geplant. Die angeforderten Funktionen sind alle implementiert worden. Auf dem RoboCup 2013 wurde der NaoDeployer ohne Probleme eingesetzt. Aufgrund der Kompatibilität zum Framework aus dem B-Human Code Release 2012, haben auch andere Teams Interesse an dem NaoDeployer gezeigt. Darum ist der Sourcecode des NaoDeployers als OpenSource Projekt veröffentlicht worden.

[1] [16] *SSH.NET Library.* http://sshnet.codeplex.com/, 2013

[2]. Ziel der Veröffentlichung ist es Teams die den NaoDeployer gern nutzen möchten die Möglichkeit zu geben ihn an ihre Bedürfnisse anzupassen und das Gesamtprojekt weiter zu entwickeln.

[2][17] Berlin, Jan Hendrik: *github of Naodeployer*. https://github.com/NaoDevils/NaoDeployer, 2013

Teil III

Verhalten

Um ein gutes Abschneiden beim RoboCup zu erreichen, ist ein gutes Verhalten der Roboter unumgänglich. Wenngleich das Verhalten nicht zu den Hauptaufgaben der PG gehört, ist die Verbesserung des Verhaltens sehr wichtig für die NaoDevils. In jedem Jahr werden neue Regeln für die SPL aufgestellt, an die es das Verhalten anzupassen gilt. 2013 ist dies hauptsächlich die Vergrößerung des Spielfeldes.

In diesem Kapitel wird die Arbeit am Verhalten beschrieben. Zunächst wird auf die Grundlagen eingegangen, die für ein funktionierendes Verhalten essentiell sind. In diesem Rahmen wird die Pfadplanung, die Anpassung der Kopfwinkel und die Berechnung des vorhergesagten Balles beschrieben.

Anschließend werden zwei neue Spielstrategien vorgestellt. Bei der Dreiecksformation positionieren sich die angreifenden Spieler in einem Dreieck, während beim Raumaufteilungsverhalten jeder Spieler für einen Bereich des Spielfeldes zuständig ist.

Schließlich wird das Verhalten in speziellen Situationen vorgestellt. Hierunter fällt das Verhalten des Torwartes sowie das Verhalten beim Elfmeterschießen.

Kapitel 7

Pfadplanung

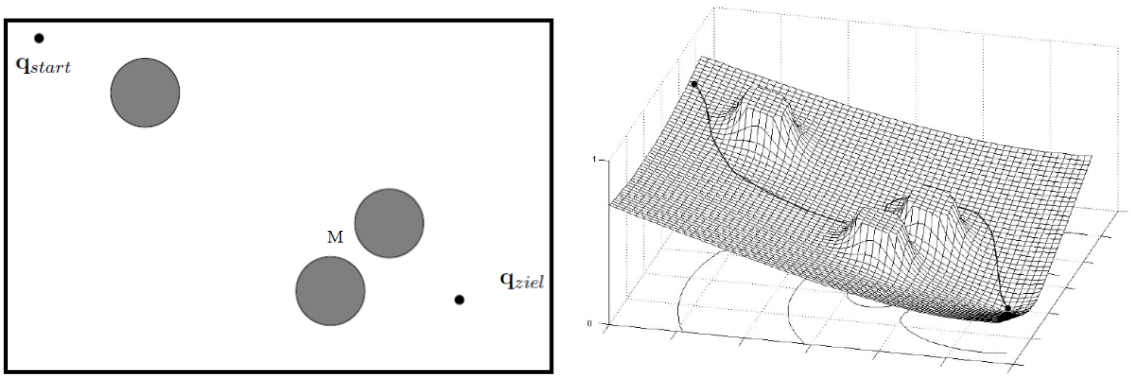
bearbeitet von: Elena Erdmann

Soll der Roboter von seiner aktuellen Position zu einem bestimmten Punkt auf dem Feld gelangen, so reicht es nicht, wenn er nur gerade auf diesen Punkt zuläuft. Er muss andere Roboter als Hindernisse erkennen und umgehen, darf niemals den Strafraum betreten und sollte sich in einer sinnvollen Richtung ausrichten. Alles das ist Aufgabe der Pfadplanung.

Eine Methode, die aus den gegebenen Feldinformationen die optimale Bewegungsrichtung des Roboters berechnet, ist die Potentialfeldmethode. Diese wird zu Beginn des Kapitels erklärt. Anschließend wird erläutert, wie aus dem Potentialfeld ein Pfad berechnet werden kann und wie dieser für den Lauf verwendet wird. Da durch die Potentialfeldmethode noch keine Rotation berechnet wird, wird weiter die Berechnung der Rotation vorgestellt.

Schließlich wird eine alternative Idee zur Pfadplanung vorgestellt, die Hindernisse durch das Ansteuern von Zwischenzielen umgeht. Es wird aufgezeigt, wie einige Aspekte dieser Idee in das Potentialfeld integriert werden können.

7.1 Potentialfeldmethode



(a) Spielsituation **(b)** Potentialfeld

Abbildung 7.1: Potentialfeld für eine Spielsituation

Grundidee der Potentialfeldmethode ist es, dass der Roboter von einem Kraftfeld gelenkt wird. Dabei zieht sein Ziel ihn an, Hindernisse stoßen ihn ab. Dafür wird jedem besonderen Punkt auf dem Feld – also dem Ziel und den Hindernissen – ein Potential zugeordnet, dass bestimmt, wie stark die Anziehung/Abstoßung an diesem Punkt ist.

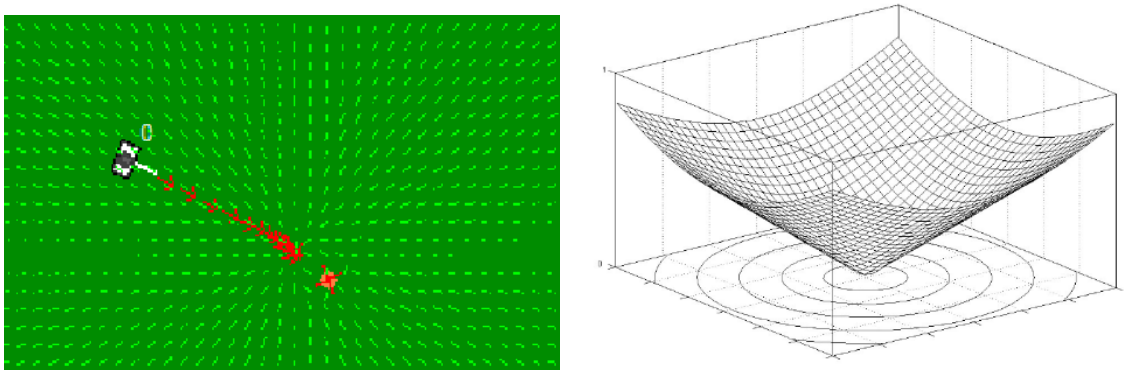
Um eine bessere Vorstellung von dieser Methode zu bekommen, kann man sich das Potentialfeld als ein gespanntes Tuch vorstellen. Am Zielpunkt wird dieses Tuch nach unten gezogen. Jedes Hindernis drückt hingegen das Tuch nach oben. Der Roboter bewegt sich nun über das Feld, wie eine Kugel über das Tuch. Durch die Schwerkraft rollt diese immer in die Richtung des stärksten Abfalles des Tuchs.

Abbildung 7.1 zeigt ein Potentialfeld für eine Spielsituation. Auf der linken Seite ist die Situation dargestellt. Der Roboter steht im Punkt q_{Start} und soll sich zum Punkt q_{Ziel} bewegen. Die grauen Kreise stellen Hindernisse dar. Auf der rechten Seite der Abbildung findet sich das zugehörige Potentialfeld. Hier ist q_{Ziel} der tiefste Punkt und die Hindernisse bilden kleine Hügel.

Der größte Vorteil des Potentialfeldes liegt darin, dass es sich mathematisch sehr gut modellieren lässt. Die verschiedenen Potentiale lassen sich als Funktionen im Abstand zu ihrem Mittelpunkt darstellen. Die verschiedenen Arten von Potentialen werden in den nächsten Unterabschnitten erläutert. Das gesamte Potentialfeld wird erhalten, indem die einzelnen Potentiale addiert werden. Schließlich soll aus dem Potentialfeld die Bewegungsrichtung des Roboters hergeleitet werden. Diese entspricht gerade der Richtung des steilsten Abstiegs des Potentialfeldes – also genau der Ableitung. Wie genau ein Pfad aus dem Potentialfeld berechnet werden kann, wird im zweiten Abschnitt dieses Kapitels erklärt.

Auch die bisherige Implementierung der Pfadplanung nutzt die Potentialfeldmethode. Allerdings wird die Potentialfunktion an diskreten Punkten berechnet und es wird eine Gaußfunktion als Potential verwendet, die sich nicht in geschlossener Form ableiten lässt. Ziel der neuen Implementierung soll also auch eine mathematisch saubere Lösung sein, die die einfache und effiziente Berechnung des Gradienten ausnutzt.

7.1.1 Attraktives lineares Potential



(a) im Simulator **(b)** als Funktion

Abbildung 7.2: Attraktives lineares Potential

Das wichtigste Potential ist das Ziel des Roboters. Dabei handelt es sich um ein attraktives, also anziehendes, Potential. Die Anziehung des Zielpunktes soll von jedem Punkt auf dem Spielfeld eine Wirkung auf den Roboter ausüben. Es ist also nicht notwendig, den Definitionsbereich der Potentialfunktion einzuschränken. Finden sich keine Hindernisse auf dem direkten Weg vom Roboter zum Zielpunkt, so soll er ohne Umweg gerade und mit konstanter Geschwindigkeit darauf zulaufen. Aus diesem Grund liegt es nahe, für das Potential des Zielpunktes eine lineare Funktion zu wählen. Da das Potential anzienhend wirkt sollte die multiplikative Konstante dabei negativ sein.

Sei (x, y) die Position des Roboters und $(x, y)_{dest}$ die des Zielpunktes in Weltkoordinaten. Weiter sei $destinationInfluence$ eine Konstante, die den Einfluss des Zieles beschreibt und

$$dist = \|(x, y) - (x, y)_{dest}\| = \sqrt{(x - x_{dest})^2 + (y - y_{dest})^2} \tag{7.1}$$

der Abstand zwischen Roboter und Zielpunkt.

Dann ist eine lineare Potentialfunktion

$$p((x, y)) = -destinationInfluence * dist. \tag{7.2}$$

Um die Kraftwirkung des Zielpunktes zu erhalten, muss schließlich die Ableitung der Potentialfunktion gebildet werden. Diese lautet

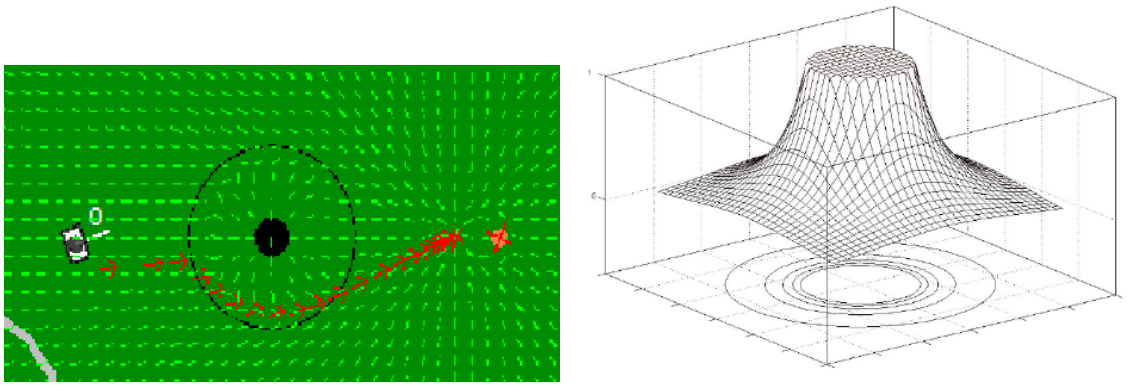
$$p'((x, y)) = -destinationInfluence * ((x, y) - (x, y)_{dest}/dist. \tag{7.3}$$

Abbildung 7.2 (a) zeigt eine Spielsituation im Simulator, bei der der Roboter ausschließlich vom Zielpunkt angezogen. Die roten Pfeile zeigen seine Bewegungsrichtung gerade auf den Zielpunkt zu. Die kleinen hellgrünen Striche sollen das Potentialfeld zeigen und sind immer in die Richtung ausgerichtet, in die der Roboter laufen würde, wenn er an diesem Punkt stehen würde. Auch sie zeigen alle gerade auf den Zielpunkt. Der zweite Teil (b) der Abbildung 7.2 ist ein Plot der zugehörigen Potentialfunktion. Den tiefsten Punkt bildet der Zielpunkt, von dort aus steigt das Potential in alle Richtungen linear an.

7.1.2 Repulsives quadratisches Potential

Hindernisse sollen den Roboter abstoßen, man bezeichnet sie als repulsive Potentiale. Im Gegensatz zum Zielpunkt, der in jedem Punkt eine Anziehung auf den Roboter auswirkt, sollen sie nur lokal wirken. Steht nämlich der Roboter so weit entfernt von einem Hindernis, dass er nicht Gefahr läuft, mit ihm zu kollidieren, so soll dieses ihn nicht in der Bewegungsrichtung beeinflussen. Um dies mathematisch zu realisieren, wird die Funktion nur in einem vorher festgelegten Einflussradius definiert, außerhalb dessen ist sie immer null.

Andererseits soll ein Hindernis, das sich sehr nah am Roboter befindet, ihn sehr stark beeinflussen, damit es nicht zur Kollision kommt. Für diese Anforderungen eignet sich eine quadratische Potentialfunktion.



(a) im Simulator **(b)** als Funktion

Abbildung 7.3: Repulsives quadratisches Potential

Sei wieder (x, y) die Position des Roboters, $(x, y)_{obs}$ die des Hindernisses, $influence$ der Einfluss und $radius$ der Einflussradius des Hindernisses und $dist$ der Abstand von Roboter und Hindernis wie oben. Dann berechnen sich das Potentialfeld und dessen Ableitung durch:

$$p((x,y)) = \begin{cases} \frac{1}{2} * influence * (\frac{1}{dist} - \frac{1}{radius})^2 & \text{für } dist < radius \\ 0 & \text{sonst} \end{cases} \tag{7.4}$$

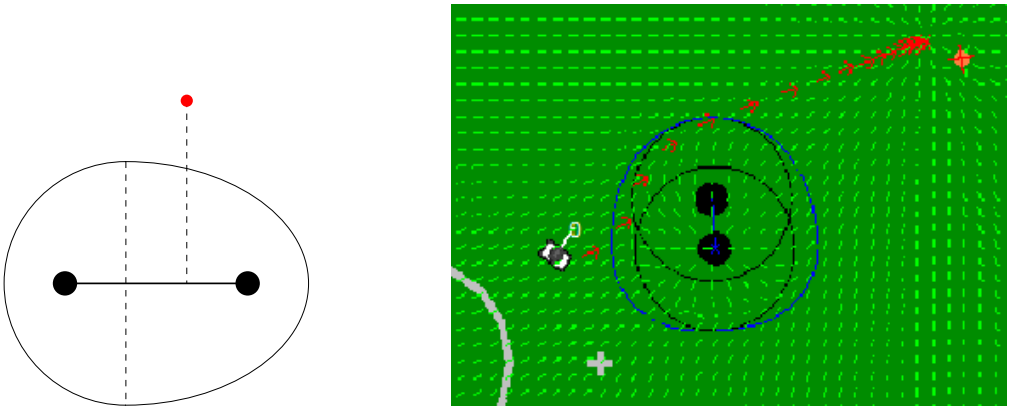
$$p'((x,y)) = \begin{cases} influence * (\frac{1}{dist} - \frac{1}{radius}) * \frac{1}{dist^3} * ((x,y) - (x,y)_{obs}) & \text{für } dist < radius \\ 0 & \text{sonst} \end{cases} \tag{7.5}$$

Steht ein Hindernis sehr nah am Ziel, so soll der Roboter trotzdem versuchen dorthin zu gelangen. Anderenfalls würde er im Spiel nicht versuchen, den Ball zu schießen, wenn ein Gegner in der Nähe ist. Um diesen Fall zu verhindern, wird eine Kollision mit dem Gegner in Kauf genommen. Um dies zu realisieren, wird bei einem nahen Gegner statt des festen Einflussradius der Abstand zum Zielpunkt verwendet:

$$radius = \min\{radius, dist(obs, dest)\} \tag{7.6}$$

7.1.3 Verschmelzen von Hindernissen

Leider kann es bei der Potentialfeldmethode mit den bisherigen Potentialen dazu kommen, dass der Roboter ein lokales Minimum der Potentialfunktion erreicht. Da er immer der Ableitung des Potentialfeldes folgt und diese in einem lokalen Minimum 0 ist, wird er dort stehenbleiben. Dieser Fall kann nur dann auftreten, wenn sich die Potentiale mehrerer Hindernisse überlagern.



(a) Vorgehen **(b)** im Simulator

Abbildung 7.4: elliptisches Potential durch Verschmelzen

Eine Möglichkeit, zu verhindern dass es zu lokalen Minima kommt, ist es, Hindernisse, die nahe beieinander stehen, zu verschmelzen. So können die Potentiale sich nicht mehr überlagern und es können keine Minima entstehen. Um dies zu erreichen müssen also die Potentialfunktionen zweier Hindernisse zu einer zusammengefasst werden.

Da der Roboter zum Einen nicht zwischen die Hindernisse gelangen soll, zum Anderen aber auch nicht viel weiter ausweichen soll als bei einem einzelnen Hindernis, liegt es nahe als Einflussradius die konvexe Hülle der Einflussradien der Einzelhindernisse zu betrachten. Da diese jeweils kreisförmig sind, entsteht so eine Art Ellipse.

Um den Roboter direkt in einer sinnvollen Richtung um die Hindernisse herumzuführen, wird zunächst der Schwerpunkt der Ellipse berechnet. Dieser soll die Stelle kennzeichnen, von dem aus der Roboter stets in der Richtung um das Hindernis geht, in der er vom Schwerpunkt aus gesehen steht. Abbildung 7.4 (a) zeigt die Berechnung des Schwerpunktes. Die beiden schwarzen Punkte repräsentieren die Hindernisse, der rote Punkt das Ziel. Erst wird das Lot vom Ziel aus auf die Verbindungsstrecke zwischen den beiden Hindernissen gefällt und anschließend die so abgeteilte Länge an der anderen Seite der Verbindungsstrecke abgetragen. Steht der Roboter also vom Ziel aus gesehen hinter dem Hindernis, so ist der kürzeste Weg um das Hindernis herum zum Ziel, immer derjenige Weg, bei dem der Roboter den Schwerpunkt nicht passiert.

Damit der Roboter das Hindernis dann auch auf die gewünschte Weise umgeht, ist das Potential am Schwerpunkt am größten. Das Potential wird durch eine quadratische Funktion beschrieben. Der Gradient zeigt so immer vom Schwerpunkt weg und lenkt den Roboter, sofern nicht andere Potentiale in die Wirkung einfließen, auf dem kürzesten Weg um die Hindernisse herum.

Abbildung 7.4 (b) zeigt eine so berechnete Ellipse im Simulator. Eine mögliche Erweiterung wäre es, auch die gewünschte Drehrichtung am Zielpunkt in die Berechnung des Schwerpunktes einfließen zu lassen.

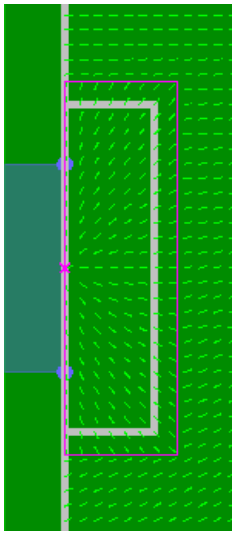


Abbildung 7.5: Potentialfeld im Strafraum

7.1.4 Umgehen des Strafraums

Die Regeln der SPL verbieten es den Feldspielern, den eigenen Strafraum zu betreten. Zunächst stellt sich hier die Frage, ob das Umgehen des Strafraums Aufgabe des Potentialfeldes oder des Verhaltens selbst ist. Sieht man diese Aufgabe als Aufgabe des Potentialfeldes, so ergibt sich eine Situation, in der der Spieler zwar in den Strafraum gehen „will“, dort allerdings vom Potentialfeld wieder herausgeschoben wird. Durch diese Konstellation wird der Spieler vor dem Strafraum hin- und hertrippeln, da er zwar den Strafraum nicht betreten kann, gleichzeitig aber auch nicht sein Ziel erreicht. Aus diesem Grund muss das Verhalten ein Ansteuern von Punkten innerhalb des eigenen Strafraumes verhindern.

Eine andere Situation ergibt sich, falls der Spieler einen Punkt ansteuern will, der zwar außerhalb des Strafraumes liegt, für den aber ein Weg durch den Strafraum geplant wird. Um dies zu verhindern, wird der Strafraum als ein repulsives Potential hinzugefügt. Da ein Betreten des Stafraums sofort zu einer Strafe führt, ist es für den Feldspieler wichtiger, diesen zu umgehen als zu seinem Ziel zu gelangen. Dementsprechend muss der Einfluss des Strafraums stärker sein als der des Zieles.

Abbildung 7.5 zeigt die Realisierung. Das Potential des Strafraums wird durch ein Rechteck repräsentiert. Dieses überdeckt den Strafraum und einen kleinen Rand darum, der dazu dient, dass der Roboter auch bei Ungenauigkeiten in der Bewegung und Lokalisierung niemals den Strafraum betritt. Das Potential wirkt innerhalb dieses Rechtecks linear abstoßend vom Mittelpunkt der Torlinie aus.

7.1.5 Umgehen des Mittelkreises

Für die SPL-Wettbewerbe 2013 wurde eine neue Regel eingeführt, die es der verteidigenden Mannschaft verbietet, direkt nach Anpfiff den Mittelkreis zu betreten. Erst nachdem

entweder der Ball bewegt wurde oder nachdem zehn Sekunden vergangen sind, wird dieses Verbot aufgehoben, der Ball wird sodann als „frei“ bezeichnet.

Eine Realisierung dieser Regel erfolgt analog zum Umgehen des eigenen Strafraumes. Verhaltensseitig darf kein Spieler der verteidigenden Mannschaft den Mittelkreis ansteuern, solange der Ball nicht frei ist. Auch hier kann es aber dazu kommen, dass ein Spieler zwar einen Punkt außerhalb des Mittelkreises ansteuert, dabei aber seinen Weg durch den Mittelkreis plant. Aus diesem Grund wird im Mittelkreis ein quadratisches Potential mit ausreichendem Einfluss angelegt.

7.2 Umsetzung im Lauf

Nachdem die Potentialfeldmethode an sich erläutert wurde, soll in diesem Abschnitt dargelegt werden, wie mit Hilfe des Potentialfeldes ein Pfad berechnet werden kann. Anschließend wird darauf eingegangen, dass es garnicht immer nötig ist, einen vollständigen Pfad zu berechnen. Stattdessen kann der Weg des Roboters auch durch eine Geschwindigkeit ausreichend vorgegeben werden.

Erfahrungsgemäß ergeben sich bei der gemeinsamen Verwendung von Instant Kick und Pfadplanung Probleme, die durch die Ausgabe als Geschwindigkeit sogar noch verstärkt werden. Eine Lösung dieser Probleme wird im letzten Abschnitt erläutert.

7.2.1 Ausgabe als Pfad

Das Potentialfeld stellt nach Hinzufügen der entsprechenden Potentiale eine Methode bereit, die den Gradienten an einem gegebenen Punkt ausgibt. Dieser berechnet sich als Summe der Gradienten der einzelnen Potentiale, die wiederum wie oben beschrieben berechnet werden.

Um aus diesem Gradienten nun einen Pfad zu erstellen, muss lediglich vom Startpunkt ausgehend sukzessive die Methode auf dem jeweils erreichtem Punkt aufgerufen werden. Dies wird so lange getan, bis das Ziel nahe genug ist oder eine feste Anzahl an Iterationen erreicht wird. Der Pfad wird durch ein Array der durchlaufenen Wegpunkte repräsentiert.

7.2.2 Ausgabe als Geschwindigkeit

Um einen gegebenen Pfad abzulaufen, wird dieser zunächst im RequestTranslator in einen Vektor aus Geschwindigkeiten übersetzt, dem der Roboter dann folgt. Daher stellt es einen großen Overhead dar, zunächst einen Pfad zu berechnen, um dieses dann in eine Richtung umzurechnen. Um sich diesen Schritt zu sparen, kann statt des Pfades direkt dieser Vektor aus Geschwindigkeiten an den RequestTranslator weitergegeben werden.

Die Potentialfeldmethode macht dies sehr einfach, da das Potentialfeld ohnehin zunächst einen Gradienten, also eine Richtung, liefert. Es muss lediglich beachtet werden, dass der

übergebene Vektor vernünftige Werte enthält. Der Roboter kann nur eine feste Geschwindigkeit für die Bewegung in eine Richtung und für die Drehung erreichen. Diese muss eingehalten werden. Andererseits ist es das Ziel der Pfadplanung, dass der Roboter möglichst schnell an sein Ziel gelangt, daher sollte die übergebene Geschwindigkeit möglichst nah an den Maximalwerten liegen, ohne sie zu überschreiten.

7.2.3 Anpassung an den Instant Kick

Die NaoDevils benutzen zum Schießen den sogenannten Instant Kick. Dieser zeichnet sich dadurch aus, dass er direkt aus dem Lauf heraus schießt. Eine langwierige Positionierung vor dem Ball ist daher nicht mehr nötig und der Roboter kann außerdem nach dem Schuss direkt weiterlaufen. Auf diese Weise ist es möglich, den gesamten Schussablauf deutlich zu beschleunigen.

Im Spiel zeigt sich jedoch, dass die Benutzung des Instant Kicks Verhalten und Pfadplanung vor neue Herausforderungen stellt. Wird der Schuss wie bisher angesteuert, so wird dem Roboter als Ziel ein Punkt kurz vor dem Ball übergeben. Um eine genaue Positionierung zu erreichen, wird der Lauf heruntergeregelt. Die Zeitersparnis durch die Einsparung der Positionierung fällt also weg. Noch problematischer ist allerdings die Auslösung des Instant Kicks. Da dieser nur durch ein Durch-den-Ball-Laufen verursacht wird, schafft es eine Positionierung vor dem Ball nicht zuverlässig, den Schuss auszulösen. Der Roboter trippelt auf der Stelle.

Ein erster Ansatz ist es, den Punkt möglichst nah an den Ball zu verschieben und die Geschwindigkeit beim Anlaufen zu erhöhen. Es zeigt sich jedoch, dass dieser Ansatz das Problem nicht zufriedenstellend beheben kann. Eine Zielansteuerung im – oder besser noch hinter dem – Ball ist notwendig.

Dies bereitet wiederum dem Potentialfeld Schwierigkeiten. Steuert der Roboter einen Punkt hinter dem Ball an, so wäre es fatal, wenn er dort direkt hinginge, anstatt sich von vorne dem Ball zu nähern. Eine Lösung dieses Problems wird in Abschnitt 7.4.2 diskutiert. Zum Anderen liegt auch im Ball ein Potential, das ein Hineinlaufen von einer anderen Seite verhindern soll. Damit der Roboter den angesteuerten Punkt hinter dem Ball auch erreichen kann, muss dieses Potential deaktiviert werden. Damit dies nur im richtigen Fall passiert, müssen Annäherungsrichtung und -winkel des Roboters genau abgefragt werden.

7.3 Drehrichtung des Roboters

Folgt der Roboter einem berechneten Pfad, so ist es sehr wichtig, dass er sich dabei in eine sinnvolle Richtung dreht. Muss er sich oft weit drehen, so dauert die Pfadverfolgung länger, da er gleichzeitig keine großen Schritte machen kann. Andererseits ist es essentiell, dass er sein Ziel und gegebenenfalls auch Hindernisse im Blick behält.

Momentan dreht der Roboter sich bei Ausführung des Pfades immer in die Richtung seines Ziels. Erst kurz (40cm) bevor er dieses erreicht fließt auch die im Zielpunkt geforderte

Drehrichtung mit ein. Dabei werden dann beide Richtungen linear gemittelt, so dass der Roboter im Ziel entsprechend der Zielrichtung steht.

Die Drehrichtung des Roboters könnte eventuell noch optimiert werden. Dazu müsste ausprobiert werden, ob etwa eine Drehung in Richtung der Hindernisse vorteilhaft ist.

7.4 Umgehen von Hindernissen durch virtuelle Zwischenziele

In diesem Abschnitt soll eine möglichen Alternative zur Potentialfeldmethode in der Pfadplanung vorgestellt werden. Diese beruht auf der Idee, möglichst oft den direkten Weg zum Ziel zu gehen und dabei Hindernisse zu vermeiden, indem virtuelle Zwischenziele neben die Hindernisse gesetzt werden. Tatsächlich findet die Idee in der Pfadplanung der NaoDevils Anwendung, wenn der Roboter sich dem Ball annähert.

7.4.1 Idee

Nicht nur beim Roboterfußball lässt sich feststellen, dass oft die einfachste Lösung die Beste ist. Auch die Pfadplanung bildet hierzu keine Ausnahme. Im Spiel zeigt sich regelmäßig, dass Teams die stur geradeaus zum Ball gehen, damit oft keine großen Nachteile gegenüber Teams, die alle Hindernisse umgehen, haben. Während ein Umgehen von Hindernissen durchaus erstrebenswert ist, sollte daher sichergestellt werden, dass dies effizient und ohne große Umwege geschieht.

Diese Überlegung führt zu der Idee, wann immer möglich den direkten Weg zum Ziel zu wählen. Nur falls ein Hindernis auf oder sehr nah an dieser Strecke liegt, soll es überhaupt in die Berechnung des Pfades einfließen. In diesem Fall soll der Roboter seitlich an dem Hindernis vorbeigehen, ohne dabei an das Hindernis zu stoßen. Um dies zu realisieren, steuert der Roboter zunächst ein virtuelles Zwischenziel mit ausreichend großem Abstand neben dem Hindernis (relativ zum Ziel) an.

Bei der Auswahl dieses Zwischenziels stellt sich die Frage, ob der Roboter links oder rechts am Hindernis vorbei laufen soll. Dafür lässt sich die Länge des Pfades durch Addition der Strecke vom Start zum Zwischenziel und von dort zum Endziel berechnen und dann vergleichen. Schwieriger gestaltet sich die Situation, falls sich auf dem neu berechneten Pfad ein weiteres Hindernis befindet. Hier muss ein weiteres virtuelles Zwischenziel eingefügt werden, was wiederum zu zwei Alternativen führt.

Obgleich das oben beschriebene Vorgehen eine exponentielle Laufzeit vermuten lässt – jeder Pfad kann sich wieder in zwei Pfade aufteilen – ist es in der Praxis leicht zu handhaben. Da einem Hindernis in einem Schritt immer komplett ausgewichen wird, gibt es immer einen Pfad (ganz außen), der keinem Hindernis zweimal ausweicht. Wendet man nun eine Breitensuche an und bricht diese ab, sobald in einer Ebene ein Pfad zum Ziel gefunden ist, so erhält man zwar nicht notwendigerweise den kürzesten, in jedem Fall aber einen sinnvollen Pfad. Durch die geringe Anzahl von Hindernissen in der RobotMap kommt es daher nicht zu großen Verzögerungen in der Laufzeit.

7.4.2 Integration im Potentialfeld

Aus Zeitgründen war es nicht möglich, die oben beschriebene Idee für den RoboCup 2013 umzusetzen. Da jedoch die Potentialfeldmethode beim Anlaufen an den Ball und insbesondere bei der Verwendung des Instant Kicks immer wieder Schwierigkeiten bereitet, wird nun zumindest in dieser Spezialsituation mit virtuellen Zwischenzielen gearbeitet.

Die Zwischenziele erfüllen dabei zwei Aufgaben. Zum Einen sorgen sie dafür, dass der Roboter bei Annäherung von der falschen Seite den Ball nicht berührt, zum anderen wird so sichergestellt, dass der Roboter sich aus der richtigen Richtung nähert um dann beim Ansteuern eines Punktes hinter dem Ball den Instant Kick auszuführen.

Die neue Idee ist dabei mit dem Potentialfeld kombiniert. Die Zwischenziele werden in der speziellen Situation vorberechnet und dann statt des eigentlichen Ziels als attraktives Potential hinzugefügt. Da so beide Berechnungen vorgenommen werden, können die Vorteile der schnelleren Laufzeit nicht genutzt werden. Insbesondere für die Benutzung des Instant Kicks werden auf diese Weise jedoch gute Ergebnisse erzielt.

Kapitel 8

Kopfwinkel nicht synchronisiert mit Bild

bearbeitet von: Michael Feininger
Ein weiteres festgestellte Problem, in den ersten Testspielen, ist die nicht korrekte Synchronisierung des Bildes mit dem Drehwinkel des Kopfes. Das Bild zeigt teilweise Abweichungen des Balles. Dieser wird auf einer anderes Position angezeigt als die Positionierung im Realen auf dem Feld. Ein still liegender Ball bewegt sich mit der Bewegung des Kopfes, dieses führt zur Fehlwahrnehmung und damit zu Ungenauigkeit bei Entscheidungen. Um dieses Problem zu lösen wurden zahlreiche Versuche durchgeführt. Bei diesen Versuchen wurden unterschiedliche Ballpositionen und Ballentfernungen vom Roboter genommen. Die unterschiedlichen Entfernungen lagen zwischen 25 und 300 cm (Abb. 8.1).

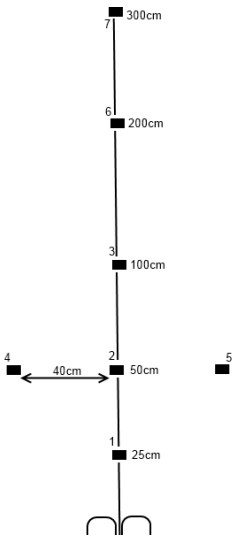


Abbildung 8.1: Versuchsaufbau

In fester Naoposition wurde der Kopf über das Feld geschwenkt und dabei der Ball gesucht. Die Erkennung des Balles wurde dann aufgezeichnet und in einem Plot wiedergegeben (Abb. 8.2).

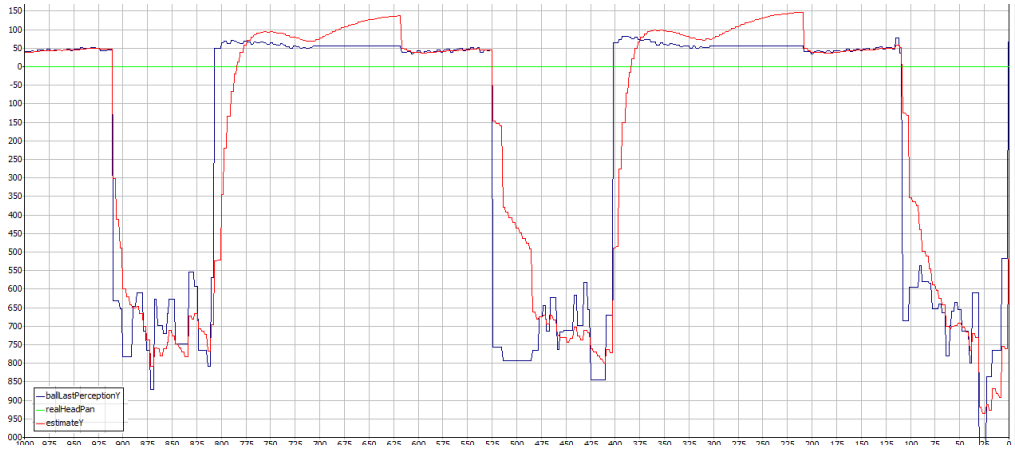


Abbildung 8.2: Auswertung des Versuchs

Nach dieser Auswertung sind in einigen Versuchen zusätzlich Abweichungen zwischen geschätzten (*Estimate Position*) und wahrgenommenen (*BallPerception*) Ballposition festgestellt worden. Aufgrund zahlreicher weiterer Auswertungen zu den Versuchen wurden die dazugehörigen Funktionen untersucht. Zunächst wurde dieses Problem zurückgestellt, im Laufe der Projektgruppe hat sich aber herausgestellt, dass diese kleinen Abweichungen sich im Spiel deutlicher auswirken als zunächst gedacht. Auf dem großen Spielfeld bedeutet diese Schwankung eine Veränderung der wahrgenommenen Ballposition von mehreren Metern. Für die Verbesserung des Roboterverhaltens ist es somit nötig geworden sich mit diesem Problem wieder zu befassen. Als Ziel der Projektgruppe für das zweite Semester wurden diese Abweichung verringert, unter anderem indem man den Rollwiderstand des Spielfeldes besser modelliert hat.

Kapitel 9

PredictedBall

bearbeitet von: Michael Feininger
Bei dem Kapitel PredigtedBall geht es um die Vorhersage des Balles. Zum Zeitpunkt der Abfrage, wird die Position berechnet an der der Ball zum stehen kommt (Abb. 9.1). Nach dem der Roboter mit der Nummer 2 den Ball auf das Tor schießt, entschließt sich der Roboter mit der Nummer 2 direkt die Wegstrecke C zur Position A zu laufen und nicht die Wegstrecke B. Diese würde der Roboter ohne der Berechnung der Halteposition des Balles laufen. Diese vorhergesagte Ballposition ist sowohl im Spiel für das schnelle Schalten der Roboter, bei der zu laufenden Wegstrecke relevant, als auch für den Torhüter sehr wichtig. Da der Torhüter sich anhand der Vorhersage in seiner Aktion entscheidet, ob und wann er sich zu einer Seite des Tores schmeißen wird. Die Formel für den Bremsweg s_B des Körpers

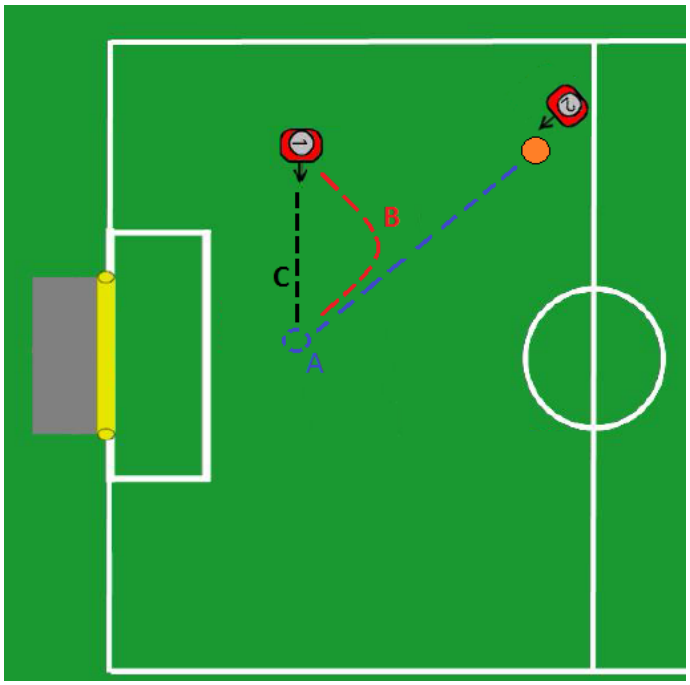


Abbildung 9.1: Optimale Wegstrecke C zum PredictedBall

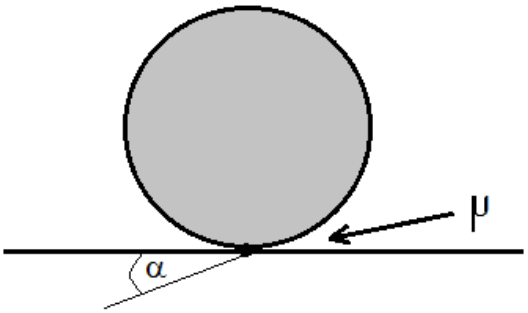


Abbildung 9.2: Reibungsfaktor μ zwischen Ball und Spielfeld

lautet:

$$s_B = \frac{V_0^2}{2a} \tag{9.1}$$

Dabei ist V_0 die Anfangsgeschwindigkeit und a die negative Beschleunigung. Die Herleitung dieser Formel wird im folgenden beschrieben.

9.1 Vorüberlegung

Zuerst setzen wir uns dafür mit den physikalischen Gesetzen auseinander. Dazu betrachten wir die kinematischen Bewegungen von Körpern unter Berücksichtigung der Reibung. In unserem Fall haben wir einen runden Körper, der eine abnehmende Geschwindigkeit hat, also eine konstante negative Beschleunigung. Die negative Beschleunigung entsteht von der Reibung zwischen den Materialien und der Gravitation. Bei unseren Fussballrobotern ist es die Reibung zwischen dem Plastikball und dem Spielfeld (Abb. 9.2). Sei der Reibungsfaktor μ , so setzt sich die negative Beschleunigung folgendermaßen zusammen:

$$a = g \cdot \mu \cdot cos(\alpha) \tag{9.2}$$

g ist hierbei die Erdbeschleunigung und α der Neigungswinkel der Fahrbahn. Das Spielfeld ist eben, also ist der Winkel 0°, und $cos(\alpha) = 1$. Der Reibungsfaktor μ hängt von dem Material der beiden Reibungskörper ab.

9.2 Herleitung

Mit der negativen Beschleunigung leitet man nun die Formel $s_B = \frac{V_0^2}{2a}$ über die Geschwindigkeit während des Bremsvorgangs her:

$$V(t) = V_0 - a \cdot t \tag{9.3}$$

Dabei ist $a \cdot t$ die Geschwindigkeit, die durchs Bremsen, also der negativen Beschleunigung, verloren gegangen ist. Die Strecke die dabei zurück gelegt wird setzt sich zusammen aus:

$$s(t) = V_0 \cdot t - \frac{1}{2} \cdot a \cdot t^2 \tag{9.4}$$

Hier ist $v_0 \cdot t$ die Strecke, die der Körper zurückgelegt hätte, wenn er keine Reibung hätte und somit ungebremst worden wäre. Der Teil $\frac{1}{2} \cdot a \cdot t^2$ ist die Strecke, die der Körper durch den Bremsvorgang weniger zurücklegt.

Bei der oben angegebenen Formel des Bremsweges kommt der Körper zum Stillstand, also ist $v(t) = 0$. In die (9.3) Gleichung eingesetzt

$$0 = V_0 - a \cdot t \tag{9.5}$$

und nach der Zeit t aufgelöst, erhält man die Bremszeit t_B:

$$t_B = \frac{V_0}{a} \tag{9.6}$$

Zuletzt setzt man die Bremszeit t_B nun in die Gleichung (9.4) ein und erhält somit die Formel für den Bremsweg:

$$s_B = V_0 \cdot \frac{V_0}{a} - \frac{1}{2} \cdot a \cdot \frac{V_0^2}{a^2} = \frac{V_0^2}{2a} \tag{9.7}$$

Hieraus berechnet sich der Bremsweg des geschossenen Balles (aus [18]). Dieser Bremsweg s_B wird als Zielposition für den Roboter gebraucht und wird zur Berechnung der Pfadplanung verwendet.

Kapitel 10

Taktiken

Durch die neuen Regeln für die Standard Plattform League des Robocups war es notwendig, das Verhalten der Roboter zu überarbeiten. Die wichtigsten Merkmale, die geändert wurden, sind eine verdoppelte Spielfeldgröße und ein zusätzlicher Spieler. Um neue Spielstrategien zu entwickeln, wurden Testspiele mit dem bisherigen Verhalten durchgeführt. Diese Testspiele und Turnierspiele mit anderen Teams wurden anhand von Videoaufzeichnungen analysiert. Nach Diskussion wurden zwei Ansätze festgelegt. Einerseits die Dreiecksformation und andererseits die Raumaufteilung, die getrennt voneinander implementiert wurden und im Folgenden genauer beschrieben werden.

10.1 Dreieckstaktik

bearbeitet von: Stefan Papon

Die Dreieckstaktik ist eine auf einen Teamleader und den Ball ausgelegte Taktik, das heißt die anderen Naos agieren anhand beider Position und deren vorherrschenden Situation. Insgesamt wird dabei mit einer Rollenverteilung gespielt, die variabel den Naos während des Spiels zugewiesen ist. Die vorhandenen Rollen sind hierbei Keeper, Ownhalfdefender, Triangledefender, Trianglesupporter und Teamleader, wobei die drei letztgenannten Rollen eine Gruppierung bilden, die nachfolgend Dreiecksnaos genannt wird. Im Zentrum der Taktik steht diese Dreiergruppe, die sich am Ball orientiert positionieren. Zunächst werden die grundlegenden Prinzipien der Taktik diskutiert. Anschließend wird auf spezielle Situationen, wie etwa das Ausfallverhalten, das heißt den Wegfall von Naos, eingegangen.

10.1.1 Rollenverteilung

Bei der Taktik gibt es insgesamt fünf Rollen, die vor dem Spiel anhand der Spielernummer des Naos verteilt werden. Im Prinzip ist zwar die Zuordnung eine subjektive Implementierungsentscheidung, doch da `Symbols` des Vorjahresverhaltens mitbenutzt werden, bei denen Player1 stehts der Keeper ist, wird der Keeper auf die erste Position gesetzt. Durch den Simulator, der ebenfalls Spielernummern für simulierende Naos hat, ist Player2 dem Triangledefender, Player3 dem TriangleSupporter, Player4 dem Teamleader und Player5 dem Ownhalfdefender initial zugeordnet. Dadurch kann das Verhalten dort besser getestet werden, ohne weitere Anpassungen zu tätigen. Die Rollenverteilung ist dabei zum Teil statisch, zum Teil dynamisch.Dies bedeutet, dass Keeper und Ownhalfdefender während des gesamten Spieles in ihrer Rolle bleiben (statisch). Die verbliebenen drei (Dreiecks-)Rollen hingegen können während des Spieles zwischen den zugewiesenen Dreiecksnaos wechseln (dynamisch). Jedoch können diese drei Naos nicht zu Ownhalfdefender oder Keeper werden. Ausnahme bei den beiden statischen Rollen ist der Wechsel in den Rollenzustand `undefined` (beispielsweise bei kurzzeitigem Ausfall während des Spiels), aus denen sie in in ihre Ursprungsrolle zurückkehren. Die jeweiligen Rollenwechselsituationen (Bedingungen für einen Rollenwechsel) werden nach der Beschreibung der einzelnen Rollen in diesem Endbericht detailliert aufgeführt. Zunächst wird jedoch auf die Ausrichtung des Dreiecks eingegangen.

10.1.2 Die Dreiecksausrichtung

Den Kern der Taktik bildet das schon angesprochene Dreieck, bestehend aus Teamleader, Triangledefender und Trianglesupporter. Hauptsächlich konzentriert sich die Taktik auf einen Nao, der aktiv zum Ball geht (Teamleader) und dabei von den anderen beiden Naos abgesichert und unterstützt wird. Aufgrund dessen müssen sich die beiden anderen Naos primär nach dem Teamleader und sekundär nach dem Ball ausrichten (Erläuterung im Abschnitt „Dreieckspositionierung“ dieses Kapitels). Der Hauptvorteil ist, dass wenn der Teamleader zum Ball geht, er in seiner Nähe zwei weitere Naos hat. Im Optimalfall gewinnt er einen möglichen Zweikampf und schießt ein Tor. In vielen Fällen wird der Ball vom Gegner irgendwo hinter den Teamleader geschossen oder rollt seitwärts vom Teamleader weg. Im ersten Fall steht in der Regel der Triangledefender hinter dem Teamleader

und kann somit den Ball, der hinter den Teamleader geschossen wird abfangen und nach vorne schießen. Rollt der Ball seitwärts, gibt es zwei verschiedene Möglichkeiten. Der Ball rollt mehr Richtung Seitenlinie (der momentan gespielten Seite in y-Richtung) oder zur Feldmitte. Wenn er Richtung Seitenlinie rollt sollte der Teamleader weiterhin optimal von den Dreien zum Ball stehen. Er ist immer noch für den Ball zuständig, außer der Ball ist zu weit hinter ihn gespielt worden, sodass der Triangledefender zum Ball geht. Bewegt sich der Ball wiederum mehr Richtung Mitte und ist der Ball näher am TriangleSupporter als am Teamleader, wird der TriangleSupporter aktiv und versucht den Ball ins Tor zu schießen.

10.1.3 Die Rollen

In diesem Abschnitt wird auf die einzelnen Rollen näher eingegangen. Dabei steht im Mittelpunkt, zu erläutern wozu diese dienen und wie sie sich abhängig vom Teamleader und Ballposition verhalten. Somit wird der Sinn der Rollenvergabe verdeutlicht und aufgezeigt wie trotz der Konzentration von drei Naos in einem Feldbereich die Verteidigung aufgebaut werden kann, dass alle Defensivbereiche abgedeckt sind und beim Ausspielen der Dreiecksformation nicht plötzlich unlösbare Unterzahlsituationen entstehen.

Teamleader

In der Regel ist der Teamleader der ballnächste Nao der Dreiecksformation (Ausnahmen siehe Abschnitt `Rollenwechsel im Dreieck`). Dabei soll er hauptsächlich die Zweikämpfe führen und den Ball ins Tor schießen. Die beiden anderen DreiecksNaos und auch der Ownhalfdefender sind nach ihm ausgerichtet. Als einziger Spieler des Dreiecks muss er sich nicht am Dreieck orientieren, da er selber der Orientierungspunkt ist. Um dies zu ermöglichen muss er den anderen Spielern seine Werte über die Teamsymbols kommunizieren. Sein Verhalten ist stark ballorientiert, da er wann immer möglich aktiv zum Ball geht. Er ist, abgesehen vom Anstoß aus der eigenen Hälfte und im Ausfallverhalten, hauptsächlich in der gegnerischen Hälfte anzutreffen. Ist er nicht in einem Ausfallverhalten, bei dem Naos aus dem Dreieck fehlen, orientiert er sich bei Ballposition in der eigenen Hälfte an der `beststrikerposition`. Der Teamleader kann außerdem durch das Setzen der `Soccer.action.behavoir` auf `support_me` dem TriangleSupporter mitteilen, dass er sich mit in den Zweikampf einmischen soll. Liegt der Ball hinter dem Teamleader, versucht er trotzdem zum Ball zu kommen. Dabei kann ein Rollenwechsel stattfindet, wenn etwa der TriangleSupporter schneller zum Ball kommt. Dies kann durch eine vom Ball abgewendete Ausrichtung des Teamleaders sogar dann passieren, wenn die Distanz zum Ball vom TriangleSupporter größer ist, aber durch die Rückwärtsbewegung des aktuellem Teamleaders dieser trotzdem nicht mehr `am_i_nearest_to_ball` ist.

TriangleDefender

Der TriangleDefender ist die wichtigste Absicherung des Dreiecks und deshalb der zweitwichtigste Nao im Dreieck. Er orientiert sich in einem Feldbereich hinter dem Teamleader.

Seine seitliche Feldorientierung (y-Richtung) ist abhängig vom Symbol `ball.field.y` (absoluter y-Wert des Balles im Feld), darf aber einen Maximalwert nach links oder rechts im Vergleich zur y-Koordinate des Teamleaders dabei nicht überschreiten. Ist seine Positionierung nicht dreieckskonform, muss sie korrigiert werden (vergleiche Abschnitt `Dreieckspositionierung`). Seine Aufgabe ist die hintere Absicherung des Teamleaders bzw des gesamten Dreiecks. Verliert der Teamleader einen Zweikampf oder landet der Ball im Inneren des Dreiecks, bringt der TriangleDefender den Ball durch einen Schuss wieder nach vorne. Dabei wird der TriangleDefender nicht automatisch Teamleader, wenn der Ball in seine Nähe kommt und er `am_i_nearest_to_ball` (in der gegnerischen Hälfte) ist. Näheres dazu folgt im Abschnitt Rollenwechselsituation dieses Kapitels.

TriangleSupporter

Der TriangleSupporter steht in direkter Nähe, seitwärts eventuell leicht nach hinten versetzt, zum Teamleader (vergleiche Abschnitt Positionierung dieses Kapitels). Die Seitswärtslage ist dabei nur effektiv, wenn er immer mittiger (auf die y-Achse bezogen) im Feld steht als der Teamleader, da er dadurch die andere Spielfeldseite (vom Mittelpunkt bis zur anderen Seitenauslinie gesehen) in der gegnerischen Hälfte abdeckt. Dorthin gesprungene Bälle kann er dann besser erreichen und in Richtung Tor schießen. Falls der Ball vor dem Dreieck ist, positioniert er sich in seiner x-Feldkoordinate mit einem x-Abstand von 500mm zum Ball, falls es seinen erlaubten Abstand zum Teamleader nicht verletzt und der Teamleader als Einziger aktiv zum Ball geht. Ist der Ball neben oder hinter ihm, versucht er zum Ball zu gelangen, bis ein Rollenwechsel ihn in den meisten nachfolgenden Situationen zum Teamleader macht oder er den Ball erreicht. Er darf Positionierungsabstände zum Teamleader verletzen, falls der Ball hinter dem Dreieck ist und in der gegnerischen Hälfte. Ist der Ball neben ihm und hat eine Koordinate, die mit seinen Positionierungsvorgaben übereinstimmen, behält er die Position. Die Positionierungsvorgaben des Dreiecks beinhalten einen Minimal- und Maximalabstand in y- und x-Richtung zum Teamleader (Ausnahme Teamleader setzt `support_me`). Er richtet sich also nach dem Teamleader aus und benutzt diverse teamsymbols für die korrekte Positionierung. Aktiv wird er in Ballnähe. Als Nachteil der Position kann angesehen werden, dass sein Verhalten wegen der Unterscheidung, ob der Ball vor neben oder hinter dem Dreieck ist, das Anfälligste für Positionierungsprobleme während des Spiels ist.

Keeper

Der Keeper ist die meiste Zeit in seinem Strafraum, kann aber gegebenenfalls aus seinem Strafraum einen zu ihn rollenden Ball ablaufen und nach vorne schießen, wenn er näher am Ball ist als der Ownhalfdefender und nah genug zum Erreichen des Balles steht. Sonst hat er noch die Möglichkeit sich hinzuschmeissen (specialAction `dive`) oder den Ball bei Schüssen zu blocken. Er verweilt bei weit entfernten Spielgeschehen mittig im Tor auf der Torlinie.

Ownhalfdefender

Beim Ownhalfdefender muss man zwei Situationen unterscheiden. Wenn der Ball in der eigenen Hälfte ist, ist er hauptverantwortlich für die Abwehrarbeit des gesamten Teams. Kommt er nicht schnell genug an den Ball, wird der Gegner zumindest einen Vorteil haben. Der Ownhalfdefender wird beim Eintritt einer Spielsituation, die das Spielgeschehen in der eigene Hälfte verlagert, den Ball als einziger Feldspieler vor sich haben und hat somit die beste Ausgangslage für die Klärung des Balles nach vorne. Dabei orientiert er sich bei Ballage in der eigenen Hälfte am Ball und nicht am Dreieck (Position des Teamleaders). Ist der Ball zu weit weg, stellt er sich in die mögliche gegnerische Schussrichtung zwischen Tor und Ball zum Blocken auf. Ist der Ball nah genug, geht er zum Ball und will ihn möglichst nach vorne schießen um die Situation zu klären. Nach Klärung soll er in die angestammte Position zurückkehren, und reagiert dann auf den zweiten möglichen Situationen, nämlich wenn der Ball in der gegnerischen Hälfte ist: Der Ownhalfdefender positioniert sich asymmetrisch zur y-Position des Dreiecks in der eigenen Hälfte halblinks oder halbrechs im Feld. Hat der Teamleader einen negativen y-Positionswert (Mittelpunkt hat den Wert Null). Ist der Teamleader, und somit auch das Dreieck, links auf dem Spielfeld und der Ownhalfdefender muss sich halbrechts in der eigenen Hälfte positionieren. Die Ausrichtung erfolgt analog bei einem positiven y-Positionswert des Teamleaders halblinks. Jedoch darf der Ownhalfdefender nicht zu weit außen stehen, da er sonst nicht schnell genug auf eine Änderung in der y-Positionierung reagieren und weniger Chancen auf einen Schussblock in Tornähe hätte, weshalb die Bezeichnungen Halblinks und Halbrechts gewählt wurde. Wie tief er in der eigenen Hälfte steht, hängt davon ab, ob das Dreieck vorne in der gegnerischen Hälfte steht, dann orientiert er sich mehr offensiv in der eigenen Hälfte. Ist das Dreieck näher an der Mittellinie als am gegnerischen Tor, orientiert er mehr in eigene Tornähe, da der Triangledefender nun eher in der eigenen Hälfte bei Spielsituationen zusätzlich aushelfen kann durch eine größere Nähe zur eigenen Hälfte.

10.1.4 Rollenwechsel im Dreieck

Wie schon erläutert, kann ein Rollenwechsel nur zwischen den drei Dreiecksrollen stattfinden. Hierbei ist zu erläutern, was genau bei einem Rollenwechsel geschieht, welche Bedingungen für Rollenwechsel existieren und welche Problematiken bei der Implementierung auftreten können. Stellt einer der drei Dreiecksnaos fest, dass seine aktuelle Rolle nicht mehr passend zur Spielsituation ist, muss eine Auswertung bei jedem der drei Naos durchgeführt werden. Dies geschieht, indem aus der `player_roledecision` (ist eine Option die auswertet, ob eine Rolle gewechselt werden soll oder eine Dreieckspositionskorrektur vorgenommen werden muss) die `triangle_three_rolechange()` aufgerufen wird.

Es gibt insgesamt zwei Bedingungsabfragen in der Option `triangle_three_rolechange()`, welche die drei Rollen jeweils lokal für den Nao bestimmen. Bei der ersten Abfrage wird derjenige, der wahrscheinlich am schnellsten beim Ball sein wird Teamleader (Die Symbolbezeichnung dafür lautet `am_i_nearest_to_ball`) . Die Verwendung von `am_i_nearest_to_ball` ist dabei korrekt, da sich der Keeper oder Ownhalfdefender in der eigenen Hälfte aufhalten. Wenn der Ball in der gegnerischen Hälfte ist, wird somit einer der Dreiecksnaos der Ballnächste (Ownhalfdefender ist ja hängend in der eigenen Hälfte bei Ball in Gegnerhälfte). Die Rollenverteilung innerhalb der Dreiecknaos erfolgt schließlich nur in der gegnerischen

Hälfte. Von den übergebliebenen zwei Dreiecksnaos, die nicht Teamleader geworden sind muss die eigene x-Koordinate mit der des jeweils Naos verglichen werden. Hat der Nao eine größere x-Koordinate, wird er TriangleSupporter. Steht er weiter hinten im Vergleich zu dem anderen übergebliebenen Nao, wird er TriangleDefender.

Aus der oberen Vorgehensweise bei der Neuverteilung ergeben sich zwangsläufig weitere Fragestellungen. Wie bringt ein Nao die anderen beiden dazu, dass sie den Rollenwechsel auch vollziehen? Dazu wird ein `tactic.role_change` definiert, welches des Nao selber setzen und bei den anderen Naos über die Teamsymbols abfragen kann. Wenn einer der Naos das `tacticsymbol` setzt, reagieren die anderen dementsprechend und führen die oben beschriebene Rollenauswertung durch. Wie wird asynchrones Rollenauswerten verhindert, welches zu Endlosschleifen bei der Rollenberechnung führen kann oder zu gleichen Rollen durch zeitversetzte Unterschiede in den Auswertungsbedingungen? In der Auswertungsoption `triangle_three_rolechange` muss jeder Dreiecksnao darauf warten, bis beide anderen Naos auch in der Option sind und führen dann erst die oben beschriebenen Abfragen der eigenen Rolle aus, wenn alle drei im ersten State der Option `triangle_three_rolechange` sind. Es wird also abgefragt, ob alle drei das tacticsymbol gesetzt haben. Damit die Auswertung nicht stehen bleibt, hat diese Abfrage ein Zeitlimit. Wie ist der beste Umgang mit dem Auftreten gleicher Rollen bei unterschiedlichen Naos? Treten doch noch gleiche Rollen auf, wird das am Ende in einem Zustand nach der Rollenauswertungsentscheidung in der `triangle_three_roledecision` nochmal behandelt. Je nach gleicher Rolle vergleicht sich dabei der Nao mit dem anderen rollengleichen Nao entweder nach kleinster x-Koordinate oder nach `am_i_nearest_to_ball`. Erst wenn alles nach der Rollenauswertung stimmt, wird nach dem Ablauf in den `target_state` gewechselt und somit die Option der Rollenauswertung verlassen.

Rollenwechselsituationen des Dreiecks

Nachdem die Bedeutung des Rollenwechsels erläutert wurde, muss nun besprochen werden, zu welchen Bedingungen ein Rollenwechsel stattfinden kann und wann es besser ist, wenn kein Rollenwechsel durchgeführt wird. Rollenwechsel finden grundsätzlich nur statt, wenn der Ball in der gegnerischen Hälfte ist oder wenn Naos ausfallen (siehe Ausfallverhalten). Bei längerer Nichtlokalisierung des Balles oder bei Lokalisierung des Balles in der eigenen Hälfte ist der Rollenwechsel aufgehoben. Geht der Teamleader beispielsweise zum Ball oder befindet sich in Ballnähe, sollte keine Rollenverteilung stattfinden. Ebenfalls darf bei der Anweisung `support_me` des Teamleaders die Rolle nicht plötzlich wechseln, wenn der TriangleSupporter auch mit zum Ball geht und dann näher ist. Solange beide zum Ball gehen bleibt das `support_me` bestehen. Eine weitere Vermeidung von Rollenwechsel wollen wir in den meisten Fällen erreichen, wenn der TriangleDefender der Ballnächste ist. Ein Rollenwechsel würde dazu führen, dass er zum Teamleader wechseln würde und beide anderen Dreiecksnaos sich, ihrer Rolle entsprechend, hinter ihm positionieren müssten und dadurch auch am Ball vorbeilaufen würden. Dies wäre ineffektiv, da der Teamleader und der TriangleSupporter stattdessen das Dreieck durch eine Rückwärtsorientierung zum Ball hin enger ausrichten könnten, wenn der Ball in der Dreiecksmitte ist und dann beide selbst mit in die Balleroberung eingreifen oder einen möglichen Schuss des TriangleDefenders verwerten könnten. Dieses Eingreifen wäre durch eine Neupositionierung nicht möglich, welche bei einem Rollenwechsel in der Situation stattfinden würde. Das Problem

des Rollenwechsels von TriangleDefender zum Teamleader in dieser Situation löst man, indem der Triangledefender bis zur Rollenwechselentscheidung eine gewisse Zeitspanne (bei uns zehn Sekunden) Zeit zur Verfügung steht, die Situation zu bereinigen, solange der Ball in seiner Nähe ist und er eine Rollenwechselentscheidung auslösen könnte. Schafft er es nicht, positionieren sich die anderen beiden Naos nach ihren Rollen zum neuen Teamleader. Die meisten Rollenwechsel finden jedoch zwischen dem TriangleSupporter und dem Teamleader statt. Neben der klassischen Situation, dass der TriangleSupporter Ballnächster ist (ohne dass `support_me` aktiviert ist), wird die Rolle auch gewechselt, wenn der Supporter in y-Richtung zwischen Ball und Teamleader ist. Dies hat den Grund, dass der Supporter mittiger (in y-Richtung) stehen soll als der Teamleader (vergleiche Abschnitt TriangleSupporter).

10.1.5 Dreieckspositionierung

(a) Die Kegelbereiche, die vom Dreieck gedeckt werden

(b) Die Positionierungsbereiche des TriangleDefenders in x-Richtung und TriangleSupporters in y-Richtung

Abbildung 10.1: Positionierung des Dreiecks. Oben Teamleader, seitlich TriangleSupporter, unten TriangleDefender)

Die Dreieckspositionierung dient der offensiven Gestaltung des Spieles, wenn der Ball in der gegnerischen Hälfte ist. Der Teamleader soll dabei zum Ball gehen und von zwei weiteren Naos als Zweikampfsabsicherung unterstützt werden. Der TriangleDefender steht dabei hinter dem Teamleader und der TriangleSupporter neben ihm, eventuell leicht nach hinten versetzt. Die Positionsseite, ob rechts vom Teamleader oder links vom Teamleader stehend, hängt von der y-Koordinate des Teamleaders ab. Ist sie negativ, hält er sich links auf und der TriangleSupporter rechts vom Teamleader, sonst umgekehrt. Allgemein haben wir durch die Dreieckspositionierung mehrere Vorteile: Die beiden anderen Roboter decken durch ihren Körper (siehe Abbildung 10.1 a) hinter und neben ihnen einen Bereich ab, wo der Ball durch einen direkten Schuss nicht hinkommen kann. Gelangt der Ball in die Nähe der beiden unterstützenden Naos im Dreieck, können sie verlorene Zweikämpfe korrigieren, das heißt der Ball geht statt in Richtung des gegnerischen Tors in die Richtung des eigenen Tors oder andere Feldseite klären sie den Ball oder werden sogar selber Teamleader. Das Dreieck steht bei weiterer Ballentfernung des Teamleaders weiter auseinander und bei engerer Ballentfernung enger beieinander. Dadurch hat man ein gutes Verhältnis zwischen

Überzahl und Zweikampfstärke auf der einen Seite (enge Lage des Dreiecks) und Blocken, Raumabdeckung und größere Ballerhaltungschance auf der anderen Seite (weitere Lage des Dreiecks). Die Positionierung ist dabei wie in der Beschreibung auszuführen. Die einzelnen Bedingungen für eine Positionierung werden aber nicht detailliert hier besprochen, da sonst der quantitative Rahmen des Endberichts zu stark überschritten wird.

Beschreibung

Die Dreieckspositionierung findet nur beim TriangleDefender und TriangleSupporter statt, immer in Relation zum Teamleader. Dabei hat der Triangledefender einen maximalen und minimalen Abstand zum Teamleader in x-Richtung und der TriangleSupporter einen maximalen und minimalen Abstand zum Teamleader in y-Richtung einzuhalten (vergleiche Abbildung 10.1b)). Dieser Minimalabstand und Maximalabstand variiert, je nachdem wie nahe der Teamleader zum Ball ist (je näher desto mehr verschiebt sich die konstant große Abstandszone in seine Nähe). Außerdem orientiert sich der x-Bereich in dem sich der TriangleSupporter aufhalten darf und der y-Bereich in dem sich der TriangleDefender aufhalten darf an jeweils zwei Bezugspunkten. Beim TriangleSupporter (x-Bereich), wenn der Ball rechts vom Teamleader ist aus einem Minimum von dem absoluten Ballabstand in x-Richtung und der x-Position des Teamleaders -500mm. Dabei ist dieser Bereich des TriangleSupporters eher ein Punkt als bei der y-Koordinate ein flächiger Bereich mit minimalem und maximalem Wert. Trifft eine Positionierungsabstandsüberschreitung vom Nao zum Teamleader ein (nächster Unterpunkt des Kapitels), kommt es darauf an, ob der Positionierungsfehler in x-Richtung oder y-Richtung geschieht. Beim TriangleDefender muss man in x-Richtung und beim TriangleSupporter in der y-Richtung innerhalb des erlaubten max/min-Bereiches einen Punkt auswählen, zu dem der Nao sich jeweils mit einem `goto`-Befehl orientiert. Beim TriangleDefender in y-Richtung wird ein Punkt angesteuert, der den y-Wert des gültigen Bereiches hat und die gleichbleibende x-Koordinate des Naos wird als x-Wert der Zielkoordinate benutzt, sodass er sich nicht nach vorne oder hinten bewegt, sondern nur zur Seite geht. Beim TriangleSupporter gilt ähnliches mit einem x-Wert, der wie oben erklärt wurde, berechnet wird und als Bewegungszielkoordinate nimmt er seinen eigenen momentanen y-Wert, sodass er sich in diesem Fall nur nach oben oder unten in dieser x-Wertpositionierung bewegt. Zusammenfassend haben der TriangleDefender und der TriangleSupporter einen Bereich, in dem sie sich gültig im Dreieck positionieren können. Beide sind abhängig vom Ball und Teamleader, aber nicht von der Position des jeweiligen anderen. Jedoch musste die eigentliche Positionierung in der `root`-Datei vom TriangleDefender und TriangleSupporter aufgerufen werden, da ein Aufruf in der Auswertungsdatei zu `goto`-Konflikten führen kann. Das `positioning`-Symbol wird in der `root` dann abgefragt. Falls es in der Auswertungsdatei gesetzt wird, blockiert es alle anderen `goto`-Möglichkeiten, damit nicht der Nao nicht einfriert durch den Aufruf von mehreren goto-Befehlen gleichzeitig. Er könnte sich nämlich bereits in eine Richtung bewegen und soll sich gleichzeitig positionieren, was durch die Setzung des `positioning`-Symbols auf `true` oder `false` serialisiert wird. Wir haben außerdem Ausnahmen, wann eine strikte Positionierung nicht erfolgt: Der Triangledefender ist selber Ballnächster bei einer Verteidigungssituation innerhalb des Dreiecks, aber es wurde noch kein Rollenwechsel vollzogen oder der Supporter reagiert auf das `support_me` bestehen vom Teamleader und darf dann die minimale Nähe missachten.

10.1.6 Verteidigungsverhalten/Defensivverhalten

Wenn der Ball in der eigenen Hälfte lokalisiert wird, handelt es sich um eine Verteidigungssituation, da der Ball einerseits in Tornähe ist und andererseits die Grundprinzipien des Dreiecks nicht mehr für die Klärung der Situation geeignet sind. Das Dreieck dient der idealen Zweikampfführung in der gegnerischen Hälfte bis maximal hin zur Feldmitte (weiche Grenze Mittellinie). Würde man das komplette Dreieck nach hinten ziehen, wäre einerseits der Vorteil der Kegelschattendeckung (vergleiche Abb. 10.1 a)) durch den TriangleSupporter und den TriangleDefender nicht mehr gegeben. Der Kegelschatten wäre dann durch die Tornähe zu klein und somit zu wenig Fläche abdeckt. Andererseits geht es in der eigenen Hälfte eher um Tordeckung als um Raumdeckung der Bereiche im Kegelschatten. Ein zusätzlicher Effekt wären dann die Anwesenheit von fünf Naos in der eigenen Hälfte, was Nachteile in der flexiblen Spielweise nach vorne zur Folge hätte, da die Umstellung auf Offensive zu langwierig wäre. Die Lösung ist deshalb ein flexibler TriangleDefender. Er verteidigt nicht nur lokal den Rückraum des Teamleaders innerhalb des Dreiecks, sondern hilft dem OwnHalfDefender in der eigenen Hälfte aus, wenn er besser zum Ball platziert ist. Die Positionierung des Ownhalfdefenders entspricht dem vorhin beschriebenen asymmetrischen Verhalten im Relation zum Teamleader. Dadurch kommt er recht gut in Ballnähe, da dieser in den meisten Fällen zentral in die eigene Hälfte rollt oder in den Außenbereich des Feldes, wo der Ownhalfdefender durch seine Asymmetrie orientiert ist (Effekt der Kegelschattierung aus Abbildung 10.1a)).

Kommt der Ball in die eigene Hälfte, auf der y-Achse im Außenbereich, wo der TriangleDefender ist, geschehen zwei Ereignisse. Der Triangledefender verlässt die zwei anderen TriangleNaos und läuft in Richtung eigenem Strafraum mit Bezug auf die y-Koordinate des Balls. Ist er sogar `am_i_nearest_to_ball` orientiert er sich direkt zum Ball und will den Ball beim Erreichen Richtung gegnerische Hälfte schießen (dribbeln war in der Taktik nicht eingebaut). Zur Erinnerung für das Verständnis des Gesamtverhaltens muss folgender Sachverhalt nochmal klargemacht werden. In der Defensive gibt es keine Dreiecksbildung und keine Dreieckspositionierung! Der Teamleader positioniert sich ab einer Balldistanz von über drei Metern nach der besten Strikerposition vorne, sonst behält er seine Position. Wenn seine aktuelle Position jedoch zu weit vom Ball entfernt ist, dass ein Schuss ihn in der `beststrikerposition` erreichen könnte, orientiert er sich einen Meter vor dem Mittelkreis. Der TriangleSupporter orientiert sich bei Ballposition in der eigenen Hälfte jeweils weiterhin am Teamleader, jedoch diagonal rückwärtsversetzt um einen halben Meter in x- und y- Richtung vom Teamleader zur Seite beziehungsweise nach hinten. Die Auswahl der Seite an der er sich orientiert hängt von der Außenfeldseite ab, in der sich der Teamleader befindet. Ist der Teamleader auf der rechten Spielfeldseite (Sichtstandpunkt vom eigenen Tor), orientiert der TriangleDefender sich links von ihm, sonst andersherum. Diese Ausrichtung soll helfen eine Balance zu finden zwischen idealem Stehen vor dem gegnerischen Tor zur Schussverwertung und Nähe zum Mittelkreis, falls einer der beiden Verteidiger doch von weiter wegschießt oder wenn Teamleader und / oder TriangleSupporter sehr nahe am Ball ist. Sind beide in Ballnähe trotz Ballposition in der gegnerischen Hälfte (Balldistanz unter einem Meter von Nao zu Ball) werden sich beide weder in Mittelpunktnähe noch in die Strikerposition begeben.

10.1.7 Ausfallverhalten von Naos

Die Behandlung des Ausfallverhaltens ist für den letzten Abschnitt wichtig, da nicht immer alle Naos zur Verfügung stehen und es Abhängigkeiten der Naos zueinander gibt. Die Lösung von Ausfällen wird an die Anzahl der Roboter und die momentanen zur Verfügung stehenden Positionen geknüpft.

Technische Details

Für das Ausfallverhalten haben wir mehrere durchnummerierte `player_roledecision` (von `five_player_roledecision` bis zum trivialen Fall `one_ player_roledecision`, für eine Auswertungsunterscheidung nach aktiver Naozahl). Jede der Dateien steht für ein Rollen- und Positionierungsentscheidungsverhalten bei einer bestimmten Anzahl von Spielern, d.h ob man positioniert oder die Rolle wechselt. Wie viele der anderen Teamspieler momentan aktiv sind, wird in einer höherstufigen xabsl-Datei ausgewertet und dadurch verwendet der Nao die richtige `role_decision`. Die Rollenwechselauswertung innerhalb des Dreiecks wird auch in drei Dateien organisiert: `triangle_three_role_decision` `triangle_two_role_decision` und `triangle_one_role_decision`, was für die Anzahl der verbliebenen Teilnehmer im Dreieck steht.

Umsetzung

In den technischen Details sahen wir, dass sowohl die Spieleranzahl als auch die Anzahl im Dreieck unterschieden wurde. Bei voller Spieleranzahl (`five_player_roledecision`) haben wir immer mit der `triangle_three_role_decision` zu tun und das Standardverhalten. Bei einem Spieler (`one_player_roledecision`) bleiben Ownhalfdefender weiter Ownhalfdefender und Keeper weiter Keeper. Handelt es sich um einen Dreiecksspieler wird er aus seiner `Native_role` in der `triangle_one_role_decision` zum Teamleader und aktiviert das Symbol `extended_role_teamleader`, was ihm ermöglicht sich in der eigenen Hälfte zu bewegen, statt sich vorne in der gegnerischen Hälfte auszurichten bei Ballposition in der eigenen Hälfte (andere Zustände im Teamleader `_rootverhalten` werden angesprochen). Er geht dabei immer zum Ball und blockt nicht, da er ursprünglich als Teamleader vorne platziert ist und von der Spielweise nicht zu einem hinten stehenden Abwehrspieler gemacht werden kann.

Bei vier verbleibenden Spieler (`four_player_roledecision`) kommt es darauf an, wo der Ausfall stattgefunden hat. Ist der Ausfall innerhalb des Dreiecks, rufen wir die `triangle_two_roledecision` auf, falls ein TriangleSupporter existiert, da dieser nun einer der anderen Naorollen annehmen muss. Die `triangle_two_roledecision` unterscheidet deshalb nur zwischen Teamleader und Triangledefender, da beide wichtiger sind als der Supporter und deren Ausfall schwerer zu kompensieren wäre. Der TriangleDefender sichert den wichtigeren Bereich als der Supporter (besser ist es hinten abzusichern als seitwärts) und der Teamleader ist der ballaktivste Nao, der zumal ein wichtiger Orientierungspunkt für andere ist. Teamleader wird der mit der kleineren Balldistanz.In der `four_player_roledecision` bleiben die Positionierungsabfragen des Triangledefenders wie

in der `five_player_roledecision`, wann er sich positionieren muss. Ownhalfdefender und Keeper bleiben in ihrer Rolle.

Wenn jedoch einer der statischen Rollen ausfällt in der Vier-Spieler-Variante rufen wir statt der `triangle_two_roledecision` die `triangle_three_roledecision` nun bei Rollenauswertungen auf. Dabei setzen wir in der `four_player_roledecision` ein Symbol `extended_role_supporter` auf true, was verhindert, dass sich der Supporter in der gegnerischen Hälfte bei Ballbesitz in der eigenen Hälfte positioniert. Stattdessen hilft er in der eigenen Hälfte den TriangleDefender aus, indem er auch dem Ball in die eigene Hälfte folgt. Außerdem ist seine Positionierung dadurch defensiver, d.h in seiner `root` (Datei wo Laufwege mit und ohne Ball definiert werden) wird bei Aktivierung sein Trianglepositionierungsbereich (Min-/Maxentfernung zum Teamleader) in x-Richtung auf eine größere Entfernung geändert. Die y-Richtung bleibt jedoch.

Bei drei verbleibenden Naos (`three_player_roledecision`) wird diesmal zwischen drei Spielern, zwei Spielern und einem Spieler im Dreieck unterschieden. Bei Dreien (`three_player_roledecision`) wird die `triangle_three_roledecision` weiterhin als Rollenverteilung benutzt, in der `three_player_roledecision` wird neben dem Standardabfragen der `five_player_roldecisions` nach Positionierungs- und Rollenwechselabfragen die beiden Variablen `extended_role_teamleader` und `extendes_role_supporter` gesetzt, sodass alle drei Naos auch nach hinten zum Ball laufen können, wenn der Ball in der eigenen Hälfte ist. Der Supporter ist im Dreieck wieder defensiver bei der Positionierung durch seine `Root`. Fallen eine der festen Rollen und ein Dreiecksspieler aus, wird die `triangle_two_roledecision` benutzt.Diese muss anfangs auswerten, damit man keinen Supporter hat und spielt dann mit einem wieder defensivmöglichen Teamleader durch das `extended_role_teamleader`-Symbol. Fallen bei Dreien zwei aus dem Dreieck aus, nutzen wir die `triangle_one_roledecision`, damit der verbliebene Nao ein Teamleader ist, jedoch lassen wir diesmal die `extended_role_teamleader` aus„ da wir in der Verteidigung noch den Ownhalfdefender und Keeper haben.

Bei zwei verbleibenden Spielern (`two_player_roledecision`) reagieren wir analog, je nachdem wo der Ausfall ist und wie viele noch im Dreieck verbleiben. Bei Keinem im Dreieck wird einfach keine `roledecision` getätigt und die Naos haben ihre beiden festen Rollen. Bei Einem im Dreieck, bei insgesamt zwei verbliebenen Spielern rufen wir wieder die `triangle_one_roledecision` auf, haben einen Teamleader, der aber nicht defensiv arbeitet (`Symbol` auf false). Bei Zweien im Dreieck bei zwei verbliebenen Spielern haben wir eine feste Rolle rufen wir wieder `triangle_two_roledecision` auf und benutzen es wieder zwischen Triangledefender und Teamleader, lassen den Teamleader aber auch nach hinten laufen lassen, da er den Triangledefender hinten unterstützen muss.

10.1.8 Realisierung und Änderungen im c++-Code

Erweiterung auf Fünf Spieler

Aufgrund der neuen Regeln besteht ein Team nun aus fünf Robotern auf dem Feld. Dies musste natürlich im Code angepasst werden. Hierzu wurde der c++-Code systematisch gesichtet und an zutreffenden Stellen ergänzt. So wurden `Enums` erweitert und Variablen für

den fünften Roboter hinzugefügt. Dies war hauptsächlich in Bereichen der Teamkommunikation notwendig, da hier von jedem Roboter im Team Daten empfangen werden. Aber auch die Namen der einzelnen Rollen mussten angepasst werden. Da wir, wie Eingangs dieses Kapitels beschrieben, eine grundlegend andere Taktik mit anderen Rollennamen implementieren, haben wir die Rollennamen unserer Taktik entsprechend angepasst und um ein Element erweitert.

Symbols

Damit Daten zwischen dem in *XABSL-* und `c++`-Code implementierten Teil ausgetauscht werden können, gibt es Symbols. Symbols sind Variablen, die man im *XABSL-* und `c++`-Code anlegen muss. Um miteinander zu kommunizieren haben wir das bestehende Symbol `soccer.behavior_action` verwendet. Da dieses Symbol ein Enum ist, kann man es sehr leicht erweitern und neue Werte hinzufügen, die dieses Symbol annehmen kann. Des weiteren wird dieses Symbol schon verteilt, will heißen, jeder Roboter weis von den anderen Robotern welchen Wert dieses Symbol gerade hat. Eine funktionierende WLAN-Verbindung ist hierbei natürlich vorausgesetzt. Die anderen Roboter werden mit dem Symbol `team.player1` bis `team.player4` identifiziert. Es sind nur Vier und nicht Fünf, da ein Roboter sich selbst nicht als Mitspieler ansieht. Um nun zu erfahren welchen Wert das Symbol `soccer.behavior_action` auf einem Roboter mit einer bestimmten Rolle hat, braucht es eine komplexe Abfrage. Um dies zu vereinfachen, schauten wir uns wiederum den `c++`-Code an und fanden die Stelle, wo empfangene Daten in die betreffenden Variablen der TeamSymbols geladen werden. Wir fügten nun für jede Rolle ein neues Symbol ein. So war es nun möglich zum Beispiel den Roboter mit der Rolle Teamleader ohne Umwege mit `theTeamleader` anzusprechen. Dies ersparte komplexe Abfragen und hielt den Quelltext etwas kompakter. Um bei Tests auf dem Spielfeld zu sehen welche Rolle der Roboter hat, erweiterten wir das bestehende LED Debuging. Die LED des rechten Auges ist nun wie folgt zu interpretieren:

- Roboter hat keinen Bodenkontakt -> Blau leuchtend.
- Keeper ist weiniger als 1m vom Ball entfernt -> Blau blinkend
- Keeper ist weiter als 1m vom Ball entfernt -> Blau leuchtend
- Ownhalfdefender -> Rot leuchtend
- Teamleader -> Rot blinkend
- Trianglesupporter -> Grün leuchtend
- Triangledefender -> Grün blinkend

Auf der RoBOW 13.1 musste möglichst schnell ein weiteres Symbol vom Typ `bool` her, welches wie `soccer.behavior_action` ebenfalls an die anderen Roboter gesendet wird. Während der Implementierung stelle sich heraus, wir benötigen zwei Symbols: Eines nannten wir `tactic.role_change`, das andere `team.role_change`. Die erste Variante dient dazu das Symbol zu setzten, dies gilt dann für den Roboter der dies gerade tut. Mit der zweiten Variante kann man der Wert eines anderen Roboters abfragen. Auf die ausführliche Beschreibung der Implementierung wird an dieser Stelle verzichtet. Im Holmes-Wiki [19] wurde ein Kapitel zur Lösung dieses Problems verfasst.

10.1.9 Testmöglichkeit im Simulator

Leider wurde erst zwei Wochen vor der RoBOW 13.1 klar, dass wir auch im Simulator sinnvoll testen können. Bis dahin hieß es immer, es sei nicht möglich bzw. viel zu langsam den Simulator mit mehreren Robotern zu starten. Glücklicherweise gab Ingmar Schwarz den Tip, dass dies unter bestimmten Umständen möglich sei. Um effizient zu testen schaltet man die Bewegung der Roboter aus und verändert ihre Positionen manuell. So ist sehr gut zu Testen, welche Rolle ein Roboter unter den derzeit auf dem virtuellen Spielfeld herrschenden Gegebenheiten einnimmt. Diese Tests haben sehr geholfen, weil wir so erst genau verstanden, was genau geschah. Bei Tests auf dem Feld hat man nicht sehr viele Möglichkeiten zu sehen was im Code gerade passiert. Dies führt dazu, dass man Vermutet statt wirklich Fehler festzustellen. Die *XABSL*-Log-Dateien sind zwar dann für die Analyse eines solchen Tests sehr hilfreich, ersetzen aber nicht die Tests im Simulator. Da der Simulator in MS Visual Studio 2008 gestartet wird, hat man des weiteren die Möglichkeit den integrierten Debugger für den `c++`-Code zu benutzen oder den Simulator anzuhalten und die *XABSL*Berechnungen Frame für Frame laufen zu lassen. So kann man genau feststellen auf welcher Datenbasis der Roboter entscheidet. Des weiteren lassen sich Situationen leichter nachstellen und die Tests verlaufen erheblich schneller.

10.1.10 Fazit Dreiecksverhalten

Das Dreiecksverhalten hatte eine enorme Komplexität entwickelt durch die Realisierungsidee. Aufgrund verschiedener Möglichkeiten sich zu Positionieren und einer großer Anzahl Bedingungen, die vor allem die Dreiecksnaos bewältigten mussten, war eine Diagnose und Fehlerbehebung extrem aufwändig. Die zusätzlichen Rollen-wechsel und bedingungen und die Lösung der Asynchronität der gleichzeitigen Abfrage der Rollenwechselbedingungen machten das Verhalten ebenso komplex, aber dafür sehr reaktiv. Das Verhalten ist deshalb reaktiv, weil die Naos auch in Situationen, wo ein anderer Nao am Ball ist, nicht einfach stehen bleiben, sondern die für sie optimale Position oder Rolle suchen, damit der Ball möglichst wieder zum Dreieck und somit in das Tor gelangt. Außerdem schafft die Dreiecksformation beim Erreichen des Balls neben den Absicherungen durch den Triangledefender und Ownhalfdefender auch ein Überzahlspiel. Trotzdem werden durch die Kegelabdeckung nicht besetzte Bereiche von den Naos abgedeckt, indem das Hinschießen in den Bereich durch Blocken unmöglich gemacht wird oder ein Nao durch seine Positionierung recht schnell zum Ball kommen kann. Der Gegner kann die Formation nicht überlaufen, außer er spielt den Ownhalfdefender aus und der Torwart reagiert nicht schnell genug, was aber bei fast allen Taktiken ähnlich wäre. Der Raum vor dem Ownhalfdefender und neben ihm ist jedoch abgesichert. Zusätzlich ist das Verhalten stark von der Wahrnehmung der eigenen Position und der Ballposition abhängig, was bei schlechter Ballerkennung zu zusätzlichen unerwünschten Verhalten führen kann. Dadurch war erst der Simulatortest zumindest erfolgreicher bei der Ideensuche der Fehlerbehebung als die analysierten Logs. Nimmt man jedoch die Idee weiter auf und wird das Verhalten stabiler in den switch-cases bei Bedingungsabfragen und bei der Ballerkennung etwas robuster, kann durch die Kegelabdeckung, die geschaffenen Absicherungen, aber auch durch die erzwungenen Überzahlsituationen diese Taktik sehr gefährlich.

10.2 Raumaufteilung

bearbeitet von: Bianca Patro, Florian Gürster

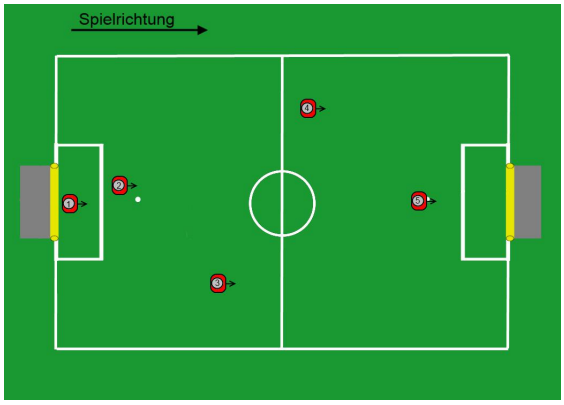
Bei der Untersuchung der Taktik vom RoboCup 2012 ist eine eher schlechte Verteilung der Roboter auf dem Spielfeld aufgefallen. Diese Verteilung könnte mit dem größeren Feld ein Problem darstellen, weil die Roboter weite Strecken zurücklegen müssen, bevor sie aktiv ins Spielgeschehen eingreifen können. Die Raumaufteilungstaktik weißt jeden Spieler einen bestimmten Aktionsraum zu um sicherzustellen, dass das Spielfeld gleichmäßig abgedeckt ist. Um den Vorteil der Nähe zum Geschehen noch weiter zu verbessern, war die Optimierung der Reaktionsgeschwindigkeit auf Ereignisse ein Ziel. Im Folgenden wird zuerst das Konzept der Raumaufteilung detailliert beschrieben, worauf die genaue Realisierung folgt. Im Anschluss folgt eine kurze Übersicht wie sich die Taktik während der Bearbeitung verändert hat.

10.2.1 Beschreibung

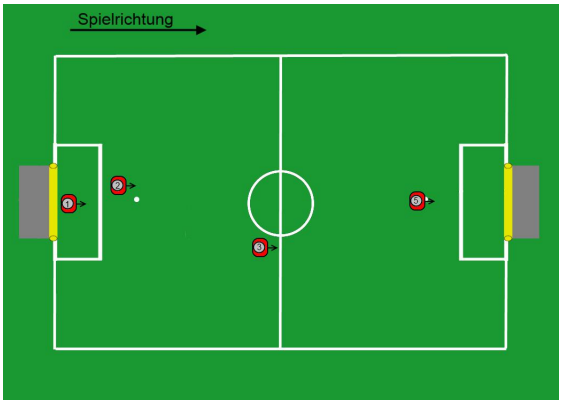
Die bisherige Taktik war auf vier Spieler ausgelegt, es gab die Rollen eines Keepers, eines Defenders, eines Supporters und eines Strikers. Der zusätzliche fünfte Spieler eröffnet neue Möglichkeiten. Die bisherigen Aufgaben der Rollen waren zufriedenstellend und die Ideen hinter den Positionen wurden für die Räume übernommen. Der Keeper positioniert sich im Strafraum und übernimmt die Aufgaben des Torhüters. Der Raum des Defenders ist vor dem eigenen Strafraum, um verteidigend zu agieren und den Spielaufbau zu gestalten. Der Raum vor dem gegnerischen Strafraum ist die Zone des Strikers, dessen Hauptaufgabe das Toreschießen ist. Bei der ehemaligen Erweiterung von drei auf vier Spieler kam der Supporter hinzu. Der Supporter bildet das Bindeglied zwischen den beiden Strafräumen. Aktiv unterstützte er den Defender oder den Striker je nach seiner aktuellen Position. Für die Einführung des fünften Spielers gab es mehrere Möglichkeiten. Ein zweiter Defender würde für eine feste defensive Ausrichtung sprechen. Ein zweiter Striker hingegen entspräche einer eher offensiven Gewichtung. Als Kompromiss wurde ein zweiter Supporter angesehen, um den neu entstandenen weiten Raum zwischen den beiden Strafräumen abzudecken. Zwei Supporter mit gleicher Aufgabe würden sich gegenseitig stören. Als Lösung wurde die Positionierung der beiden Supporter als Flügelspieler vorgesehen, die sich das Spielfeld in der Vertikale teilen. Damit beide Supporter nicht gleichzeitig den Defender oder den Striker unterstützen, bekommen sie auf ihrem Flügel eine defensive bzw. offensive Grundausrichtung. Als Folge haben sich in der Raumaufteilung die Positionen des Keepers, des Defenders, des defensiven Supporters, des offensiven Supporters und des Strikers ergeben. Bei voller Mannschaftsstärke ergibt sich eine Rautenaufstellung, die bei Wegfall eines Spielers zum Dreieck wird (Abb. 10.2).

Die Räume eines Robotes teilen sich in eine *Main Area* und eine *Shared Area* auf. Die *Main Area* ist der Hauptaufenthaltsort eines Roboters, diese wird von der *Shared Area* umschlossen, in der der Roboter mehr interagieren soll. Durch eine vorherige Festlegung dieser Räume kann eine vollständige Abdeckung des Spielfeldes erreicht werden (Abb. 10.3). Einzelne Zonen überlappen sich unterschiedlich stark um die aktivsten Zonen bei einem Spiel abzudecken.

Standardmäßig begibt sich der Roboter auf eine optimale Position in der *Main Area*. Die optimale Position ist abhängig von der zugeordneten Spielposition. Für den Keeper be-



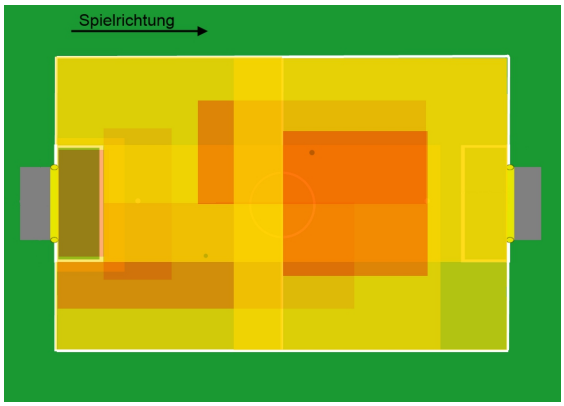
(a) Skizze der Aufstellung bei fünf Spielern



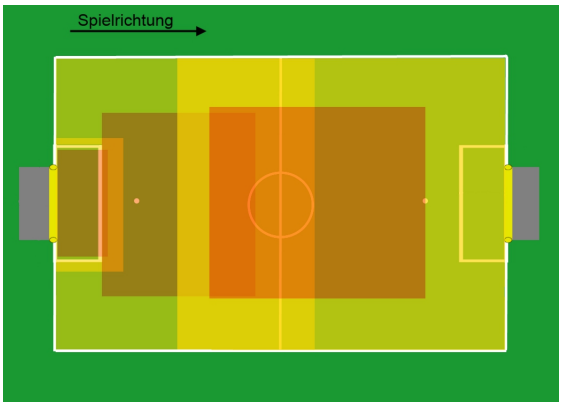
(b) Skizze der Aufstellung bei vier Spielern, ohne offensiven Supporter

Abbildung 10.2: Skizze der Grundaufstellung bei fünf und vier Spielern auf dem Feld

findet sich die optimale Position in der Mitte der Torlinie. Der Defender steht zwischen Strafstoßpunkt und Strafraum versetzt zum Keeper, um einerseits das Tor besser abzudecken und anderseits dem Torhüter nicht die Sicht zu versperren. Die optimalen Positionen der Supporter befinden sich im Zentrum ihrer *Main Area*, wo sie um bis zu 90 Grad gedreht zur Richtung der Spielfeldmitte stehen. Die genaue Ausrichtung ist abhängig von der Position der y-Koordinate des Spielers. Die Drehung der Supporter wurde eingeführt, um bei dem vertikalen Spielverlauf Richtung Tor den Ball weniger im toten Winkel hinter den Robotern zu haben. Der Striker nimmt eine optimale Position für den Torabschluss in seiner *Main Area* ein.



(a) Überlagerung der Areas bei allen Spielern



(b) Überlagerung der Areas bei drei Spielern (Keeper und beide Supporter)

Abbildung 10.3: Überlagerung der Areas mit zwei verschiedenen Spieleranzahlen (Gelb: *Shared Area*, Orange: *Main Area*)

Im Laufe des Spiels kann es zu Situationen kommen, in denen der Roboter die Position des Balles nicht mehr kennt. Sobald der Roboter den Ball eine längere Zeit nicht gesehen und keine Ballpositionen der anderen Roboter erhalten hat, geht er zur Ballsuche über. Die Suche nach dem Ball startet von der optimalen Position des Roboters und umfasst mit

der Zeit einen größeren Raum. Sobald der Ball gefunden wurde und die Position verifiziert werden konnte, wird in das normale Spielverhalten übergegangen.

Die Roboter gehen zum Ball, wenn sie am nächsten zum Ball stehen. Bei der Berechnung des Abstandes wird nicht einfach die Distanz zwischen den beiden Punkten ausgerechnet, sondern es wird der mögliche Weg beachtet. Auf die Länge des Weges zum Ball hat nicht nur die reine Entfernung Einfluss, sondern auch eine die und eventuelle Hindernisse, wie andere Spieler. Die einzige Ausnahme ist der Keeper, der auch zum Ball geht, wenn der Ball sich in seinem Strafraum befindet. Der Keepter ist der einzige Spieler der sich im eigenen Strafraum aufhalten darf. Falls der Roboter sich in seiner *Main Area* befindet und den Ball kontrolliert, dann dribbelt er Richtung Tor. Sobald der Spieler den Ball in der *Shared Area* kontrolliert, passt er ihn nach vorne. Falls er nah genug am Tor ist, versucht er einen Torschuss. Als besonderes taktisches Verhalten schirmt der Defender den Ball ab, wenn der Ball im eigenen Strafraum liegt, damit der Gegner nicht schnell zu einer guten Position für den Torabschluss kommen kann.

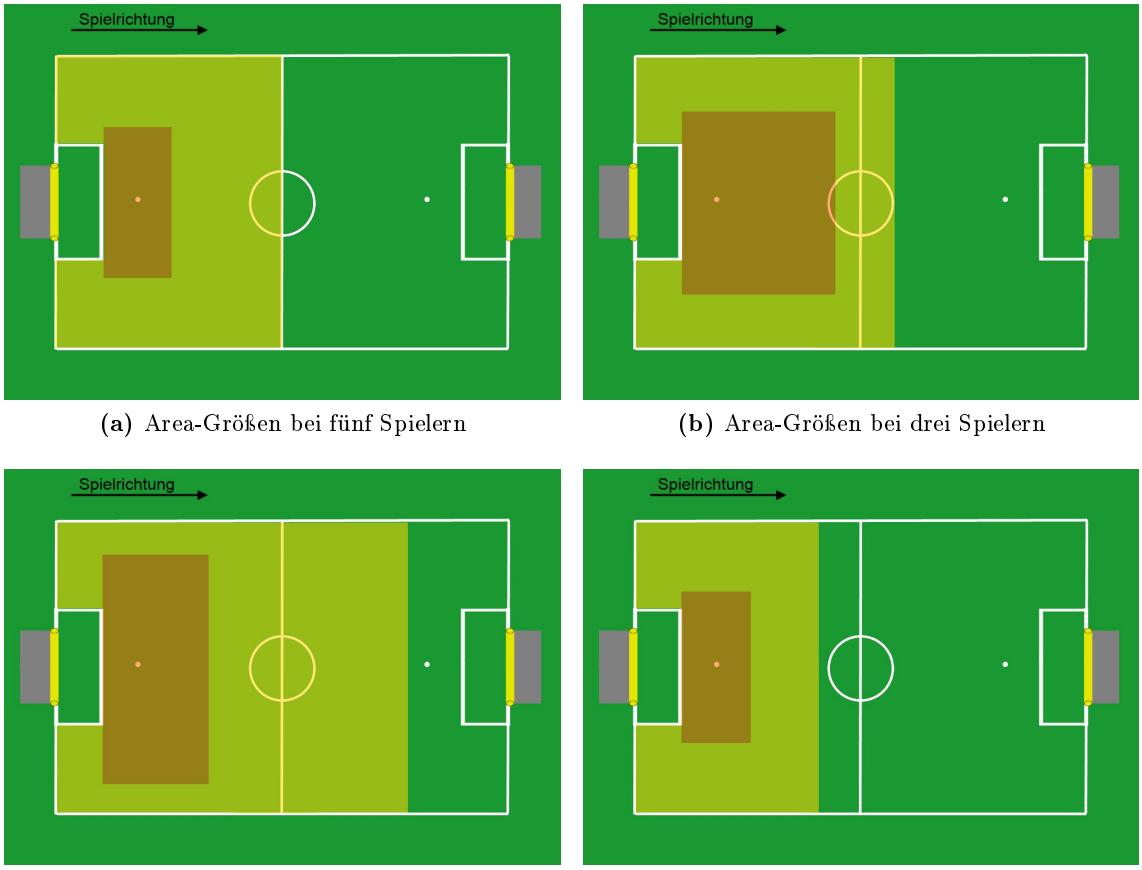
Im Spielverlauf kann sich die Anzahl der beteiligten Spieler verändern, was keine großen Auswirken auf diese Taktik hat. Für jede Spieleranzahl wurde vorher eine Area-Verteilung definiert und somit bedient ein Spieler bei abnehmender Spieleranzahl einen größeren Raum (Abb. 10.4). Bei einem einzelnen Feldspieler, wird das gesamte Spielfeld von der einen Area Kombination abgedeckt. Als Ausnahme für die Regelung für einzelne Spieler bestehen die Räume des Keepers und des Defenders, damit die Möglichkeit eines Torschusses für den Gegner erschwert wird.

Ein weiterer Vorteil der festen Positionen ist, dass die Roboter, falls sie nicht in der Nähe des Spielgeschehens sind, stabil stehen bleiben und somit gute Beobachtungsdaten über Ball und Positionen geben können. Das zuverlässige Wissen über die Ballposition verhindert Fehlverhalten und minimiert die Ablenkung durch die Ballsuche.

Bei den Spielen kann es immer zu Situationen kommen, in denen das WLAN-Signal schwankt. Bei einer WLAN-Störung werden oft veraltete Pakete empfangen, die Folgen sind veraltetes Wissen über die Mitspieler und möglicherweise ein Schwanken der Anzahl der verbundenen Mitspieler. Die Raumaufteilungstaktik sollte gegen solche Störung recht immun sein. Durch die schwankende Anzahl der bekannten Mitspieler, verändert sich die optimale Position. Die optimalen Positionen für verschiedene Spieleranzahlen liegen aber nicht so weit auseinander, dass weite Strecken zurück gelegt werden müssen. Die Roboter warten nicht mehr nur an einer Stelle, sondern bewegen sich zwischen den optimalen Positionen Bei der Entscheidung ob ein Roboter der Nächste zum Ball ist, wird die Ballnähe der anderen Teammitglieder berücksichtigt. Wegen den veralteten Ballinformationen könnten sich zwei Roboter zuständig fühlen und gleichzeitig zu unterschiedlichen Teambällen gehen.

Die schnellen Entscheidungen wurden durch eine geringe Entscheidungstiefe realisiert. Die Entscheidungen sind wenig verschachtelt und auch möglichst allgemein, um das Risiko für Deadlocks in Spezialfällen auszuschließen.

Auch die Aufstellung für den Kickoff wurde an das neue größere Feld angepasst. Als anstoßendes Team darf nur ein Spieler im Mittelkreis sein und der Rest der Mannschaft nimmt eine gute Position in der eigenen Spielfeldhälfte ein. Beim gegnerischen Anstoß wird auch



(a) Area-Größen bei fünf Spielern

(b) Area-Größen bei drei Spielern

(c) Area-Größen bei zwei Spielern

(d) Area-Größen bei einem Spieler

Abbildung 10.4: Area-Größen des Defenders für vier verschiedene Spieleranzahlen (Gelb: *Shared Area*, Orange: *Main Area*)

der Anstoßkreis als nicht erlaubte Zone für die Positionierung ausgeschlossen. Die Spieler positionieren sich in der Nähe ihres optimalen Punktes in ihrer *Main Area*, solange es eine nach den Regeln valide Position ist. Eine Ausnahme macht beim Kickoff der Striker. Ein schnelles Reagieren auf den Anpfiff hat Vorrang. Sollte die Strikerposition nicht besetzt sein, rückt der nächst offensivere Spieler auf diese Position nach.

Im nächsten Abschnitt wird die genaue Umsetzung der Raumaufteilungstaktik beschrieben.

10.2.2 Realisierung

In diesem Abschnitt werden die Implementierungsdetails der Raumaufteilung geschildert. In der alten Taktik gab es dynamische Rollen, welche im Laufe des Spiels – mit Ausnahme des Keepers – gewechselt haben. Dadurch waren einige Options zur Rollenverteilung für verschiedene Anzahlen von Spielern sowie zur Synchronisation notwendig. Diese Dynamik sorgte in manchen Fällen für Probleme, wenn zum Beispiel das WLAN ausfiel und die Rollen nicht synchronisiert werden konnten. Da bei der Raumaufteilungstaktik alle Spieler

anstatt Rollen eine feste Spielposition haben, konnte dieser Bereich komplett aus dem Code entfernt und dadurch die Anzahl der Options reduziert werden.

Die Positionen der Spieler werden beim Aufsetzen der Roboter durch die Spielernummer zugeordnet. Dabei gilt die folgende Zuordnung:

- 1 – Keeper
- 2 – Defender
- 3 – SupporterDef = defensiver Supporter
- 4 – SupporterOff = offensiver Supporter
- 5 – Striker

Wenn weniger als fünf Roboter aufgesetzt werden, dann lässt sich durch die Spielernummer angeben, welche Positionen im Spiel besetzt sind, indem die entsprechenden Nummern vergeben werden. Für die Umsetzung der Raumaufteilung wurde eine Config-Datei `areas.cfg` angelegt, in welcher für alle Spieler und alle Kombinationen von Spieleranzahlen die Koordinaten für *Main* und *Shared Areas* aufgelistet sind. Für die Verarbeitung der Koordinaten sowie die Berechnung von Positionen wurden die `AreaSymbols` eingebaut. Dort werden alle Areas aus der Config-Datei eingelesen und in zwei zweidimensionalen Arrays von Rechtecken gespeichert. Eine *Area* ist immer durch zwei Koordinaten-Paare repräsentiert, welche in einem `Rectangle` gespeichert werden. Das ermöglicht einen einfachen Zugriff auf die einzelnen Koordinaten. Die Werte der Config-Datei werden in den beiden Arrays `mainAreas` und `sharedAreas` gespeichert. So lassen sich für alle Spieler die entsprechenden *Areas* als `Rectangle` und darüber die einzelnen Koordinaten abfragen.

Zunächst wurden mit den AreaSymbols die folgenden Symbole zur Abfrage in *XABSL* bereitgestellt:

- `float input area.opt_position_x`
- `float input area.opt_position_y`
- `float input area.opt_rotation`
- `bool input area.is_robot_in_main_area`
- `bool input area.is_robot_in_shared_area`
- `bool input area.is_ball_in_main_area`
- `bool input area.is_ball_in_shared_area`

Dabei handelt es sich um `Input-Symbols`, was bedeutet, dass die Werte in *XABSL* ausgelesen, aber nicht gesetzt werden können. Die Symbole `area.opt_position_x` und `area.opt_position_y` geben für jeden Roboter Koordinaten für eine optimale Positionierung an. Für die Berechnung der optimalen Positionen gibt es für die einzelnen Rollen private Methoden. Dabei fließt auch die aktuelle Spieleranzahl mit in die Berechnung der Positionen ein. Der Wert von `area.opt_rotation` gibt für die beiden Supporter und den Striker eine Rotation auf dem Spielfeld an, die an der Zielposition eingenommen werden soll. Der Keeper und der Defender sollten initial bei der Positionierung immer in Richtung des gegnerischen Tores gedreht sein. Im späteren Verlauf wurde ein neuer Torwart entwickelt, welcher auch eine andere Positionierung und Rotation erhält. Diese Werte wurden aber trotzdem über die gleichen Methoden gesetzt (siehe Kapitel 11).

Die vier booleschen Symbole sind für Abfragen, ob sich der Roboter gerade in seiner *Main* oder *Shared Area* befindet oder ob der Ball sich gerade in der *Area* des Roboters befindet. Für diese Abfragen wird überprüft, ob sich die Koordinaten des Roboters, beziehungsweise des Balls, innerhalb des Rechtecks befinden, welches durch die entsprechende *Area* aufgespannt wird. Diese Koordinaten erhält man aus den entsprechenden Representations `RobotPose` beziehungsweise `BallModel`. Mit Hilfe der Symbole wurden die Ideen des Verhaltens zunächst in *XABSL* umgesetzt.

In der *XABSL*-Implementierung ist `pre_initial_state` die Root-Option, mit der das Verhalten startet. Von dort aus wird `start_soccer` aufgerufen, welches die Kontroll-Optionen `body_control`, `official_button_interface`, `display_control`, `head_control` und `sound_control` aufruft. In der `body_control` läuft die Reaktion auf die GameStates ab. Im GameState `Ready` positionieren sich die Roboter auf den vorgesehenen Kickoff-Positionen, welche in den `TacticSymbols` definiert sind. Das eigentliche Spiel startet mit der Option `playing`. Hier werden je nach Rolle des Roboters unterschiedliche `*_root`-Options aufgerufen, welche für jeden Roboter das Spielverhalten enthalten.

Für den Keeper wurde dort noch das alte Verhalten verwendet, welches nur in Bezug auf die Koordinaten des neuen Feldes angepasst wurde. Für den Defender werden in `defender_root` vier Fälle unterschieden. Wenn der Ball länger als sechs Sekunden von keinem Teammitglied gesehen wurde, wird die Option `search_for_ball` aufgerufen, welche die Ballsuche einleitet. Ansonsten wird überprüft, ob der Ball im eigenen Strafraum liegt. Diesen Strafraum darf nur der Keeper betreten, also kann kein eigener Feldspieler den Ball dort erreichen. Deswegen stellt der Defender sich vor den Ball und blockiert so den Weg für gegnerische Spieler. In der dritten Abfrage wird getestet, ob der Defender der Nächste zum Ball ist. Ist er der Nächste, so geht er zum Ball. In allen anderen Fällen begibt sich der Defender auf seine optimale Position. Für die anderen Feldspieler laufen die Entscheidungen ähnlich ab, wobei die anderen Feldspieler kein Sonderverhalten haben, wenn der Ball im eigenen Strafraum ist. Dort werden die Fälle „Ball nicht gesehen“, „nächster zum Ball“ und „sonst“ abgefragt und dementsprechend behandelt. Im „sonst“-Fall positionieren sich die Spieler auf den optimalen Positionen, welche in den `AreaSymbols` berechnet werden. Damit sind diese Options sehr ähnlich, weswegen sie bei der Portierung aufs neue Framework zusammengefasst und vereinheitlicht wurden.

Für die Annäherung zum Ball sowie die Ballkontrolle wurde diese Vereinheitlichung bereits in *XABSL* vorgenommen. Daraus ist die Option `player_control_ball` entstanden, welche für alle Feldspieler gültig ist. Dort erfolgt zunächst eine Annäherung an den Ball mittels `go_to` mit der optimalen Goalkick-Position als Zielkoordinaten. Wenn der Ball erreicht ist, wird unterschieden, ob der Ball in der *Main Area* des Roboters ist oder nicht. Ist der Ball in der *Main Area*, so dribbelt der Roboter bis zum Rand seiner *Main Area*, bevor er schießt. Das Dribbeln ist ebenfalls über ein `go_to` realisiert. Am Rand der Area oder wenn der Ball von vornherein nicht in der *Main Area* ist, schießt der Roboter den Ball mit Hilfe des InstantKicks weg.

Insgesamt werden wenige Entscheidungen in *XABSL* getroffen und die Verschachtelungstiefe der Options wurde möglichst gering gehalten.

Die Entscheidung, wer zum Ball geht, ist davon abhängig, wer der Nächste zum Ball ist. Diese Information liefert das Symbol `tactic.am_i_nearest_to_ball`. Im Zuge einer

Anpassung der `TacticSymbols` an die neue Feldgröße sowie an fünf Spieler wurde auch dieses Symbol untersucht und einige Fehler wurden festgestellt und behoben. Im Folgenden wird die aktuelle Funktionsweise des Symbols kurz erläutert. Zunächst wird die euklidische Distanz zwischen dem Roboter und dem Ball ermittelt. Auf diese Distanz werden Werte addiert, wenn der Roboter sich erst drehen muss, um zum Ball zu laufen, sich beim Ball erst drehen muss, um aufs Tor zu schießen oder wenn Hindernisse im Weg sind. Das Symbol lässt sich über vier Parameter einstellen:

- `targetDistanceRobotRotFactor` – Winkel, um den der Roboter sich zum Ball drehen muss, wird mit diesem Faktor multipliziert
- `ballDistRobotToTargetAngleFactor` – Der Winkel, um den der Roboter sich am Ball zum Tor drehen muss, wird mit 1,5 potenziert und durch diesen Faktor geteilt
- `ballDistanceObstacleFactor` – Faktor zur Gewichtung von Hindernissen
- `distanceHysteresis` – Die Hysterese wird zur Stabilisierung der Entscheidung verwendet.

Jeder Roboter berechnet für sich seine modifizierte Balldistanz, also die Distanz unter Einbeziehung von Rotation und Hindernissen. Diese modifizierte Balldistanz steht über die `TeamMateData` jedem Roboter zur Verfügung. Um das Symbol `tactic.am_i_nearest_to_ball` zu setzen, überprüft jeder Roboter für sich, ob er eine geringere modifizierte Balldistanz hat, als die anderen Roboter. Wenn ein Robter bereits auf dem Weg zum Ball ist, so wird für die anderen Roboter die Hysterese hinzu addiert, um ständiges Flackern zu vermeiden. Bei dem Vergleich der Distanzen werden Roboter, die gerade penalized sind, gerade aufstehen oder hochgehoben sind, sowie Roboter, die den Ball lange Zeit nicht mehr gesehen haben, nicht beachtet.

Nach den GermanOpen wurde komplett auf ein neues Framework umgestellt, was eine Ablösung der Verhaltenssprache *XABSL* durch *CABSL* mit sich brachte (siehe Kapitel 5). Dementsprechend musste die Raumaufteilungstaktik auf diese neue Sprache portiert werden. Da im Vergleich zur alten Taktik die Anzahl der Options bereits deutlich reduziert worden war, waren in dem Bereich nur wenige Dateien umzuschreiben. Auch die Berechnungen für die optimalen Positionen und Rotationen waren bereits in `c++` geschrieben und machten daher nur geringen Aufwand. Durch Wegfall der *XABSL*-Symbols musste an der Struktur gearbeitet werden, weil alle Informationen nun direkt aus den Representations genommen werden.

Die Options-Struktur blieb größtenteils erhalten, nur die `*_root`-Options wurden zu einer Option `player_root` zusammengefasst. Des Weiteren wurde das Dribbeln nicht mehr implementiert, so dass Roboter, die am Ball sind, nun direkt schießen. Aufwändig gestaltete sich beispielsweise die Portierung von Funktionen aus den `TacticSymbols`.

Zwischen GermanOpen und RoboCup lagen nur insgesamt acht Wochen. Als die Entscheidung fiel, auf das neue Framework zu wechseln, waren es nur noch vier Wochen bis zum RoboCup. Das ist für die Behebung von Fehlern, eine komplette Portierung und nach Möglichkeit die Entwicklung neuer Features wenig Zeit. Parallel gab es eine weitere Taktik, für welche bereits grundlegende Funktionalitäten wie Kopfbewegungen, Schüsse oder auch Berechnungen von Schusspostionen portiert worden waren. Um bereits gemachte Arbeit nicht doppelt zu machen und auch, um den Portierungsaufwand klein zu halten, wurden einige der bereits implementierten Funktionalitäten übernommen. Da allerdings

viele dieser Implementierungen speziell auf die andere Taktik ausgerichtet waren, kam es zu Kompatibilitätsproblemen und Manches musste doch neu gemacht werden. Aufgrund dieser Probleme wurde viel Zeit benötigt, um die Raumaufteilungstaktik wieder auf einen qualitativ gleichwertigen Stand wie vor der Portierung zu bekommen. Später wurde diese Taktik dann eingestellt.

Im nächsten Abschnitt wird ein Fazit zur Raumaufteilungstaktik gezogen, welches auf verschiedene Tests und Entwicklungsentscheidungen eingeht.

10.2.3 Fazit

Während der Entwicklung wurden mehrere Testspiele durchgeführt, deren Erkenntnisse in der Weiterentwicklung der Taktik berücksichtigt wurden. Einige wichtige größere Tests und deren Einfluss werden nun beschrieben.

Als erstes Testspiel mit vollständiger Implementierung kann man ein Testspiel bei dem Workshop RoBoW 13.1 im Frühjahr 2013 sehen. Mehrere Roboterfußball-Teams waren in Dortmund zu Gast und die Raumaufteilung hat einen 10-minütigen Drei gegen Drei Test gegen die Rekordweltmeister B-Human aus Bremen gespielt. Das Zusammenspiel der Roboter funktionierte sehr gut. Als Beispiel kann man eine Szene nennen, in der ein Spieler im Mittelfeld den Ball bis zum Rand seiner *Shared Area* gedribbelt hat und dort an den Striker übergeben hat. Dieser Test wurde mit 2:1 gewonnen. Im Nachhinein betrachtet war dies der beste Zustand in dem sich dieses taktische Verhalten befand. Die Teamleitung wollte die Raumaufteilungstaktik als taktisches Verhalten für die GermanOpen 2013 nutzen.

Bis zum Turnier wurden gewisse Feinheiten in der Zuständigkeitsverteilung und im Umgang mit dem bisherigen Roboterverhalten optimiert. Bei den GermanOpen 2013 gab es drei verschiedene Versionen des taktischen Verhaltens der Raumaufteilung, die im Folgenden genauer beschrieben werden. Kurz vor den GermanOpen im April 2013, sollte ein neues Roboterverhalten unter anderen für die Pfadplanung, die Kopfbewegung, die Lokalisierung und Ballannäherung übernommen und eingebaut werden. Die Testreihen auf dem heimischen Feld verliefen ohne Probleme. Ende April 2013 in Magdeburg bei den GermanOpen wurde kurz nach der Ankunft ein erneuter Test gegen B-Human mit fünf gegen fünf Spieler vereinbart. Dieses Testspiel wurde mit 0:6 innerhalb von zehn Minuten verloren. Bei dem Test gab es erhebliche Probleme. Zwei der eigenen Spieler reagierten nicht auf die WLAN-Signale und wurden deshalb vor dem Spiel vom Feld genommen. Zwei weitere Roboter, unter anderem der Torwart, konnten sich durch eine falsche Kalibrierung nicht auf dem Feld lokalisieren und haben deshalb nicht aktiv ins Spiel eingegriffen. Nach dem Spiel wurde von der Teamleitung entschieden, ein alternatives Verhalten eines Mitarbeiters für das Turnier zu benutzen.

Parallel dazu wurde eine überarbeitete Version der Raumaufteilung erstellt, die auf der Version von der RoBoW 13.1 basierte und nur Änderungen enthielt, die ausführlich getestet waren. Auf das Dribbeln wurde verzichtet, weil die Justierung auf den dortigen Boden zeitlich zu knapp geworden wäre. Diese Version wurde in einem Test über eine Halbzeit gegen den späteren Gegner SPQR aus Rom benutzt, welcher mit 1:1 ausging. In diesem Test hat sich gezeigt, dass es noch nötig war, die Parameter und Kalibrierungen auf das Mageburger Feld einzustellen.

Weil die meisten Teammitglieder mit der Behebung eines Problems des alternativen Verhaltens beschäftigt waren, wurden in einer dritten Version des Verhaltens die Parameter und Kalibrierungen des Alternativverhaltens übernommen. Kurz vor Beginn des Spiels gegen SPQR wurde das alternative Verhalten repariert und die Roboter wurden mit beiden aufgesetzt, so dass die Teamleitung bei Bedarf wechseln konnte. Begonnen wurde mit der Raumaufteilung. Früh im Spiel stürzte ein ballführender Spieler, nach dem Aufstehen hatte er leicht die Orientierung verloren und das ungedeckte Tor knapp verfehlt. Im darauf folgenden Konter sind eigene Roboter wegen Lokalisierungsproblemen stehen geblieben und es ist ein Gegentor gefallen. Nach dem einen Gegentreffer wurde von der Teamleitung auf das alternative Verhalten umgestellt. Das Spiel endete mit einer 0:2 Niederlage.

Nach den GermanOpen wurde das neue Framework (siehe Kapitel 5) eingeführt. Die Portierung sollte bis zum RoboCup 2013 Ende Juni innerhalb eines Monats geschehen. Das alternative taktische Verhalten war zum größten Teil bereits portiert, weshalb dieses als Grundlage zur Portierung verwendet wurde. Auf die Portierung der Spezialfälle für die einzelnen Rollen der Feldspieler wurde verzichtet. Ein neuer Keeper wurde eingeführt. Um die taktischen Verhalten nach der Portierung zu testen, wurde auf heimischem Feld ein Testspiel angesetzt. Nachdem durchgängig 24 Minuten gespielt wurde und einige Roboter wegen Überhitzung und Verschleiß ausfielen, hat das Team mit dem andere Verhalten ein Tor geschossen und das Spiel wurde beendet. Die Teamleitung hat sich nach dem Spiel dazu entschlossen, die Raumaufteilungstaktik nicht weiter zu verfolgen. Der Torhüter wurde später in die Turnierverhalten für den RoboCup 2013 übernommen (siehe Kapitel 11).

Der Torwart wurde zwar als Teil der Raumaufteilungstaktik entwickelt, wurde aber später in das genutzte Verhalten für den RoboCup integriert und auch in der Drop-In-Challenge verwendet. Aus diesem Grund folgt die Beschreibung des Torwarts im nächsten Kapitel und nicht als Teil dieses Verhaltens.

Kapitel 11

Torwartverhalten

bearbeitet von: Bianca Patro, Heinrich Pettenpohl

Die Verwendung des alten Torwartverhalten zeigte sich als nicht sinnvoll, da dieser zu langsam reagiert und nach dem Hinschmeißen große Lokalisierungsprobleme hat. Aus diesem Grund wurde ein neuer Torwart entwickelt, der schnellere Entscheidungen trifft und des Weiteren neben Hinschmeißen eine andere Blocktechnik zur Verfügung hat.

Im Folgenden wird zunächst die Positionierung des Torwarts beschrieben. Anschließend werden die Blockmöglichkeiten erläutert und die neue Entscheidungsstruktur vorgestellt. Im letzten Abschnitt wird der Torwart dann anhand des RoboCups evaluiert.

11.1 Positionierung des Torwarts

Der Torwart ist der einzige Spieler, der den eigenen Strafraum betreten darf. Für die Positionierung war es wichtig, dass der Torwart das Tor gut abdeckt und sich nicht zu oft umpositioniert, da aus dem Lauf heraus keine SpecialActions ausgeführt werden können. SpecialActions sind Abfolgen von Gelenkstellungen, welche innerhalb von einer bestimmten Zeit angefahren werden und so bestimmte Bewegungsmuster wie zum Beispiel Blockpostionen ermöglichen. Welche Möglichkeiten es zum Blocken gibt wird im nächsten Abschnitt beschrieben.

Um das Tor gut abzudecken, sollte sich der Torwart innerhalb gewisser Grenzen am Ball ausrichten. Dabei ist die X-Position statisch und befindet sich kurz vor der eigenen Grundlinie (siehe Listing 11.1). Dies ermöglicht dem Torwart die vordere Linie des Strafraums zu sehen und mit Hilfe der Ecken des Strafraums seine Lokalisierung zu bestimmen.

Listing 11.1: Optimale Torwartposition auf der X-Achse

```
double AreaSymbols::getOptKeeperPosX(){
  return (double)(fieldDimensions.xPosOwnGroundline + 150);
}
```

Die Y-Position des Torwarts ist abhängig von der Ball Position (siehe Listing 11.2). Sollte der Ball zu weit weg sein, stellt er sich in die Mitte des Tores. Dies verhindert, dass der

Roboter sich auf Grund der schwierigen Ballerkennung in der Ferne ständig neu positioniert. Außerdem bleibt dem Torwart bei Fernschüssen genügend Zeit den Ball richtig zu erkennen und ihn dann zu blocken. Wenn der Ball länger nicht mehr gesehen wurde, positioniert sich der Torwart mittig vom Tor. Das kann z. B. passieren, wenn der Ball wieder in die Richtung des Gegners geschossen wird, allerdings von Spielern die Sicht blockiert wird.

Sollten die beiden ersten Fälle zutreffen, wird die Y-Position zwischen Ball und Tormittelpunkt berechnet. Das Koordinatensystem, welches im Spielfeldmittelpunkt seinen Ursprung hat, wird soweit verschoben, dass der Ursprung im Tormittelpunkt liegt. Dazu wird auf die beiden X-Werte die Grundlinienposition addiert. Danach wird der Winkel des Balls zum Ursprung berechnet

$$tan(\alpha) = ball.y/(ball.x + xPosOwnGroundline) \tag{11.1}$$

und mit diesem Winkel dann die Y-Position des Torwarts

$$optKeeperPosY = tan(\alpha) * (optKeeperPosX() + xPosOwnGroundline). \tag{11.2}$$

Dies kann zusammengefasst dargestellt werden als

$$\begin{aligned} optKeeperPosY = (ball.y/(ball.x + xPosOwnGroundline))* \\ (optKeeperPosX() + xPosOwnGroundline) \end{aligned} \tag{11.3}$$

Allerdings kann diese Rechnung auch dazu führen, dass die Y-Position ins Unendliche wandert, wenn der Ball auf der Höhe des Torwarts liegt. Dementsprechend wird danach die Y-Position auf die Breite des Tores beschränkt.

Listing 11.2: Optimale Torwartposition auf der Y-Achse

```
double AreaSymbols::getOptKeeperPosY(){
  double timeSinceLastSeen =
     BallSymbols::getTimeSinceLastSeen();
  Vector2_D<double> ball = BallSymbols::getBallField();

//Ball is far away
  if((ball-Vector2_D<double>
     (fieldDimensions.xPosOwnGroundline, 0)).abs() > 4000)
    return 0;

//If the ball wasn't seen  for a long time, position to the
   center
  if(timeSinceLastSeen > 8000)
  {
    return 0;
  }
//If the ball was seen, calculate the Point between the Ball
   and the Goal to place yourselve there
  else
  {
//transform Coordinatesystem to 0/0 at the Groundline/midGoal
    ball.x += fieldDimensions.xPosOwnGroundline;
```

```
    if(ball.x == 0) //No 0 division!
      ball.x = 1;

    double newLocatorY = ((ball.y/ball.x) *
       (getOptKeeperPosX()+fieldDimensions.xPosOwnGroundline));

//Only return calculated y if it is inside the goal Dimensions
    if(newLocatorY > fieldDimensions.yPosLeftGoal-400)
      return fieldDimensions.yPosLeftGoal-400;
    else if(newLocatorY < fieldDimensions.yPosRightGoal+400)
      return fieldDimensions.yPosRightGoal+400;

    return newLocatorY;
  }
}
```

Neben der eigentlich Positionierung muss der Torwart sich auch zum Ball drehen, um diesen immer im Blickfeld zu haben. Dementsprechend ist die Rotation auch vom Ball abhängig (siehe Listing 11.3). Da für die Ansteuerung der Wert in Grad vorliegen muss, wird erst die Torwartrotation und der Winkel zum Ball in Grad umgerechnet. Danach werden die beiden Winkel addiert, um die Rotation in Feldkoordinaten zu bekommen. Dabei sollte der Winkel nicht größer sein als 45 Grad, damit immer das Spielfeld im Blickfeld bleibt.

Listing 11.3: Optimale Torwartrotation

```
double AreaSymbols::getOptKeeperRot(){
  double ballAngle = toDegrees(ballModel.estimate.getAngle());
  double robotAngle = toDegrees(robotPose.rotation);
  //BallAngle is relativ + robotAngle to get field rotation
  double result = robotAngle + ballAngle;

  if(result > 45)
          result = 45;
  else if (result < -45)
          result = -45;

  return result;
}
```

Hat er seine optimale Postion erreicht, wechselt er in den Zustand `look_ball` wo er nur auf den Ball guckt und sich nur dann umpositioniert, wenn seine neue optimale Postion um gewisse Schwellwerte von seiner aktuellen Postion abweicht. Dadurch wird verhindert, dass der Roboter immer auf der Stelle hin und her tänzelt, da die Lokalisierung immer leicht springt und nie fest auf einer Stelle bleibt.

11.2 Blockmöglichkeiten

Das alte Torwartverhalten hat zum Blocken nur die SpecialAction `goalkeeperDefend` mit der er sich auf den Boden schmeißt. Diese SpecialAction nutzt die volle Länge des Roboters aus und deckt damit beim Blocken einen großen Raum ab. Der Nachteil ist, dass der Roboter recht lange braucht, um wieder aufzustehen und dabei häufig die Lokalisierung verliert. Dadurch kann es passieren, dass obwohl der Ball initial geblockt wurde, der Gegner dennoch ein Tor schießt, bevor der Torwart wieder in der Lage ist, in das Spielgeschehen einzugreifen.

Als Vorüberlegung für eine andere Blockmöglichkeit wurden Videos von verschiedenen Tunierspielen analysiert, um sich diesbezüglich die Taktiken anderer Teams anzuschauen. Neben dem Dive wurde häufig noch ein Ausfallsschritt oder eine breite Hocke beobachtet, die je nach betrachtetem Team unterschiedlich stabil und effektiv ist. Ein ausgeschalteter Nao wurde in die verschiedenen Postionen gebracht, um diese auf ihre Stabilität und Blockweite zu untersuchen. Abschließend wurde entschieden, einen Ausfallsschritt zu implementieren, bei dem der Nao runter geht und ein Bein ausstreckt (siehe Abbildung 11.1).

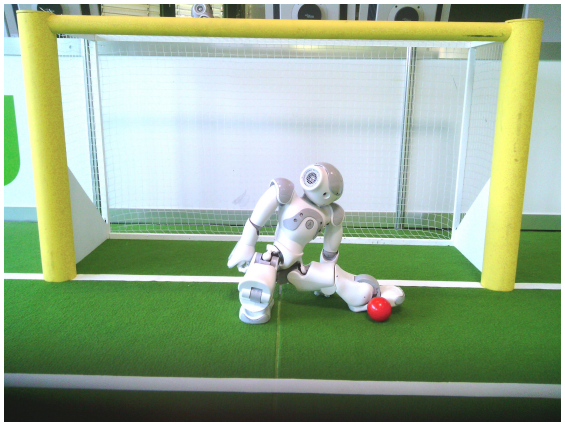


Abbildung 11.1: SpecialAction `keeperBlockLeft`

Dieser Bewegungsablauf wurde in der SpecialAction `keeperBlockLeft` umgesetzt. Das „blockLeft“ im Namen stammt daher, dass die Bewegung so implementiert ist, dass das linke Bein ausgestreckt wird. Über die Angabe von `mirror=true` im Aufruf ist die SpecialAction spiegelbar und es wird das rechte Bein ausgestreckt.

11.3 Neue Entscheidungsstruktur

Neben den Möglichkeiten den Ball zu blockieren wurde auch die Entscheidungsstruktur angepasst. Dabei blockiert der Torwart den Ball nur, wenn er selbst innerhalb des Strafraums steht, der Ball maximal 200 cm entfernt ist und innerhalb von 3 Sekunden die Torlinie erreicht. Je nachdem, ob der Ball zwischen 10 und 50 cm am Torwart vorbei geht, führt dieser dann die SpecialAction `keeperBlockLeft` aus. Bei 50 bis 1000 cm führt er die SpecialAction `keeperBlockLeft` aus.

Nicht nur die Entscheidung beim Blocken wurde überarbeitet, sondern auch der Schusswinkel wurde verbessert. So schießt der Torwart nicht mehr wie früher den Ball in Richtung des gegnerischen Tors, sondern in die Richtung, die für ihn den kleinsten Laufweg bedeutet. Dies ist meistens geradeaus, solange er nicht in Richtung seines eigenen Tores steht. Selbst wenn der Ball weit am Rand liegt, bleibt der Torwart mit dieser Taktik zwischen Ball und Tor. Im alten Verhalten hätte der Torwart den Weg Richtung Tor frei gemacht. Des Weiteren läuft er solange zum Ball, bis entweder der Ball weiter als 100 cm von ihm entfernt ist oder ein Mitspieler näher als 40 cm zum Ball ist. Dies soll verhindern, dass der Torwart sich umdreht und wieder zum Tor läuft, obwohl der Ball nur wenige Zentimeter weit gerollt ist. In diesem Szenario hätte der Torwart keine Möglichkeit den Ball zu halten, da er ihn nicht sehen kann.

11.4 Evaluation des Torwarts

Während der GermanOpen zeigte sich bei 3 offiziellen Spielen und 12 Torschüssen auf unser Tor, dass das alte Torwartverhalten nur einmal den Ball halten konnte. Dies geschah eher zufällig während eines Positionswechsels. Insgesamt hält der Torwart also 8,3 % der Bälle.

Zum RoboCup wurde das Verhalten wie oben beschrieben verbessert und während des RoboCups eingesetzt. Dabei konnte der Torwart bei 7 offiziellen Spielen und 33 Torschüssen insgesamt 17 Torschüsse halten. Das heißt, er hält 51,5 % der Bälle. Dementsprechend müssen die Parameter zum Blocken noch effektiver angepasst werden. Teilweise wurde auch auf Bälle reagiert die nicht an der Torlinie angekommen wären.

Blockmöglichkeit	Anzahl
Stehen bleiben	1
Zum Ball laufen und schießen	12
`goalkeeperDefend` (Schmeißen)	2
`keeperBlockLeft`	2

Tabelle 11.1: Häufigkeit der eingesetzten Blockmöglichkeiten während des RoboCups

Wie häufig dabei die einzelnen Blockmöglichkeiten eingesetzt wurden ist der Tabelle 11.1 zu entnehmen. In beiden Fällen, in denen die SpecialAction `keeperBlockLeft` eingesetzt wurde, konnte der Ball zwar initial gehalten werden, wurde bei dem einen Mal aber durch die Aufstehbewegung und beim anderen Mal vom Gegner ins Tor geschossen. Dies zeigt, dass die SpecialAction bezüglich ihrer Aufstehbewegung überarbeitet werden muss. Nach der SpecialAction `goalkeeperDefend` konnte nur einmal der Ball effektiv weg geschossen werden. Auch landet diese SpecialAction meistens unsanft auf dem Roboterarm. Das Landen auf dem Arm sorgt dafür, dass der Roboter mal nach hinten und mal nach vorne umkippt oder seitlich liegen bleibt. Wenn er nach vorne kippt besteht die Gefahr auf den Ball zu fallen, dies wird vom Schiedsrichter bestraft. Beim seitlichen Liegen bleiben steht der Torwart manchmal nicht mehr auf, da er sich dauerhaft „im Fall“ befindet. Am besten funktionierte das zum Ball laufen und schießen, da viele Teams meistens nicht direkt bis ins Tor schießen konnten. Das Team B-Human mit ihren starken und schnellen Fernschüssen war dabei natürlich eine Ausnahme.

So zeigt der RoboCup, dass das Torwartverhalten schon wesentlich stärker ist, als das alte Verhalten. Allerdings ist es noch nicht vollends ausgereift. Denn trotz der 17 gehaltenen Schüsse konnten nur 9 Angriffe effektiv gehalten werden. Bei den restlichen 8 Schüssen ging der Ball kurze Zeit später ins Tor. Dementsprechend hat der Torwart noch Verbesserungspotential.

Das nächste Kapitel stellt das Elfmeterverhalten vor.

Kapitel 12

Elfmeterverhalten

bearbeitet von: Stefan Papon

Im Elfmeterverhalten geht es um die Umsetzung eines Elfmeterschießens aus Sicht des Stürmers und des Torwarts. Wenn es nach regulärer Spielzeit unentschieden steht, müssen die Teams in der K.O.-Runde gegeneinander Elfmeter schießen. Die wichtigste Änderung zu den Elfmeterschießen in den Vorjahren ist im Regelwerk, dass der Stürmer nur noch einmal berühren darf, das heißt bei Berührung wird dies als Schuss gewertet und nachfolgend ist auch keine Nachschussmöglichkeit gestattet. Dies erfordert bei einem Zeitrahmen von einer Minute pro Schuss einen präzisen Anlauf zum Ball und einen guten Standpunkt bis zum Schuss, da der Ball optimal mit einer Berührung getroffen werden muss. Aufgrund dessen handelt es sich beim Elfmeterschuss um ein komplexes, parameterabhängiges Problem, da alle eingesetzten Parameter den Erfolg des Schusses bedingen und fehlende Parameteranpassungen bei anderen Spielfeldern, anderen Naos oder nach Neukalibrierungen dazu führen können, dass der Ball nicht mit einer Berührung in die optimale Torrichtung geschossen werden kann.

12.1 Schützenverhalten vom Elfmeterpunkt

(a) Die einzelnen Phasen des Elfmeterschießens aus Sicht des Strikers

Abbildung 12.1: Elfmeterphasen

Das Schützenverhalten wird in verschiedene chronologisch ablaufende Ausführungsabschnitte unterteilt. Dies zeigt die Grafik 12.1, in der die Unterteilung nochmal verdeutlicht wird. In der `Initialisierung` geht es um den Zeitpunkt, zu dem der Schütze auf dem Startpunkt steht und sich noch nicht in eine Feldrichtung bewegt hat. In der `weiten Anlaufphase` geht es um den ersten Bewegungsabschnitt, bei dem der Nao sich vom Initialpunkt Richtung Ball bewegt, jedoch noch keine Feinjustierung für den Schuss stattfindet. Ziel ist bei dieser Phase, möglichst nah in Ballrichtung zu kommen, indem die Bewegung jedoch nur in X-Richtung, das heißt in Torrichtung stattfindet, die Y-Richtung und Rotation nicht weiter verändert werden (abgesehen beim Rutscheffekt auf dem Feldteppich beim Bewegen). Die nächste Phase ist die `Schussjustierungsphase`. Dort wird eine Rotation und eine Positionierung in Ballnähe so gesucht, dass für die jeweils gewählte Torschussrichtung der Nao so steht, dass er möglichst in die Torseiten schießt, ohne dabei ins Aus oder an den Pfosten zu schießen. Jedoch darf auch nicht so mittig ins Tor geschossen werden, dass der Torwart es bei zentraler Torpositionierung einfach hat den Schuss abzublocken. Die letzte Phase ist die `Schussphase`, wo es um den Schuss, die richtige Beinwahl, aber auch um die Stabilität des Naos während des Schusses geht. Zusätzlich zu dem Hauptelfmeterverhalten, welches in diese Phasen unterteilt werden kann, gibt es sozusagen alternative Handlungsabläufe, die bei unerwünschten Handlungsausführungen (Erläuterung im Abschnitt) oder zeitlich verzögertem Verhalten aktiviert werden und zu einem Schussabschluss innerhalb der geforderten Minute führen, welcher jedoch nicht so präzise in der Ausführung und Stabilität ist, wie der normale Ablauf (siehe auch Unterabschnitt `Die Absicherung und der Beschleuniger`). Zu dem einzelnen Phasen braucht der Schütze eine passende Headcontrol, da bei falscher Headcontrol der Ball nicht richtig wahrgenommen werden kann. Da viele Abläufe über den relativen Ballabstand berechnet werden, führt eine fehlerhafte Headcontrol zu unerwünschten Effekten, wie etwa einer falschen Schusswinkelberechnung oder im schlimmsten Fall zu einem Nichtbewegen. Außerdem kann eine vorzeitige Ballberührung entstehen durch zu langes Geradeauslaufen, ohne dass die Justierung oder Schussphase erreicht wurde und somit der Ball ungewollt vom Nao überlaufen wird.

12.1.1 Die Initialisierung

In der Initialisierungsphase befindet sich der Nao immer in einer ähnlichen, aber nicht identischen Anfangsposition (zumindest praktisch). Der Schiedsrichter stellt nämlich den Nao per Augenmaß auf die Anfangsposition, das heißt der Einlauf wird nicht durch den Gamecontroler geregelt. Idealerweise soll der Nao in einem Abstand von (-1000,0) zum Ball versetzt starten und so gedreht sein, dass der Blickwinkel zum Ball keine Rotationsabweichung hat. Trotzdem kann in der Regel sowohl eine leichte Positionsabweichung als auch eine Rotationsabweichung eintreffen. Die Rotationsabweichung wird vor der Schussjustierung im Abschnitt Schusswinkel durch die Ausrichtung an der Tormitte als Rotationspunkt behandelt. Der Ball und die Tormitte sind nach Felddefinition auf den gleichen Y-Koordinaten und wenn der Nao sich dem Ball nah genug nähert, gleicht die Rotation zur Tormitte die Fehlrotation nahezu exakt aus, da beide nach der späteren Korrektur in etwa auf der gleichen Blickrichtung des Naos liegen. Die Initialisierung zieht sich dabei durch mehrere `Options`. Initial wird auch in der `Infrastructure` der Startpunkt des Naos auf den Absolutpunkt (1700,0) gesetzt, da der Elfmeterpunkt bei dem Feldkoordinatenpunkt(x=2700,y=0) liegt und der Start nach Regelwerk (siehe [splrules2013]S.12 f.) einen Meter vor dem Elfmeterpunkt ist. Auch wenn eine Abweichung durch die Schieds-

richterpositionierung entstehen kann, weis der Nao dadurch schon in etwa wo er sich wahrnehmen soll. Anfänglich bedeutet die Initialisierungsphase, dass der Nao den Ball durch `search_for_ball` sucht (auch wenn er in der Regel den Ball vor sich sehen sollte). Danach wechselt er in die Option `PenaltyStrikerControlBall` von wo aus initial eine `localize` -Headcontrol für zwei Sekunden ausgeführt wird. Damit wird zusätzlich sichergestellt, dass neben dem Ball auch die Umgebungslinien und die Torpfosten zur Orientierung wahrgenommen werden (Bei `localize` dreht der Nao den kopf von links nach rechts). Nachdem die initiale Positionierung und Orientierung abgeschlossen ist, beginnt die `weite Anlaufphase`. Die verstrichene Zeit setzt sich in der Initialphase zusammen aus der benötigten Zeit für die `bodycontrol`, die dafür zuständig ist, dass der Nao aktiviert ist (Aktivierungsbedingungen für einen möglichen Gamecontrolereinsatz inklusive), der Zeit für `search_for_ball`, die in der Regel sehr gering oder bei direkter Wahrnehmung null Sekunden beträgt, und der Lokalisierung mittels `localize`, die etwa zwei Sekunden dauert. Nach Testläufen bleiben dabei etwa noch 50 Sekunden von den benötigten 60 Sekunden über. Dies stellt den bisher gemessenen worst-case dar, falls die Ausführung gelingt.

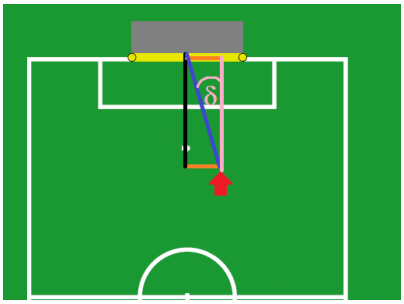
12.1.2 Die weite Anlaufphase

In der `weiten Anlaufphase` geht es um die erste Bewegung Richtung Ball, die jedoch nur geradeaus gerichtet verläuft. Der Nao orientiert sich nachdem er in der Initialphase den Ball erkannt hat nun vordergründig am Ball. Dies geschieht in dieser Phase, aber auch bei der Positionierung in der Justierungsphase relativ zum Ball. Dies hat den Vorteil, dass das Hinlaufen zum Ball nicht von der absoluten Feldposition abhängig ist, die bei der Wahrnehmung des Naos stark abweichen kann und je näher der Nao am Ball ist, desto besser funktioniert die relative Annäherung. Bis auf einen leichten Rutscheffekt wird er sich von der Anfangsposition geradeaus Richtung Ball orientieren. Dies wird neben dem ballorientierten `goto-Befehl` auch mit `localBehaviorControlOutput.behaviorData.soccerState = BehaviorData::goToBall` sichergestellt (diesen Befehl gibt es in ähnlicher Form auch im alten Framework). Beim Lauf zum Ball muss die `Headcontrol` auf `ball` oder `direct` gesetzt werden. Im Idealfall wird mit Ball der Ball vom Nao in der Phase, aber auch in der Schussjustierungsphase, dauerhaft fixiert. Falls auf einem vorher nicht getesteten Feld die Headcontrol in der weiten Anlaufphase sich aufgrund Wahrnehmungsfehler nicht auf den Ball fixiert, wird deshalb alternativ `direct` genommen, da der Nao dann geradeaus guckt. So bleibt er in der weiten Anlaufphase trotzdem auf den Ball fixiert und erst bei der Justierung sollte wieder auf `ball` gewechselt werden, da dort die Kamerawahrnehmung des Balles besser funktioniert (wie gesagt, nur bei Störungen auf `direct` wechseln). Ist Stürmer etwa in X-Richtung 20 cm vom Ball entfernt wird zur Schussjustierungsphase übergegangen, wo unter anderem auch anhand der Torlokalisierung dann der Rotationswinkel angepasst wird, falls es beim weiten Anlauf Abweichungen durch die relative Bewegung gab.

12.1.3 Schussjustierung und der Schusswinkel

Die Schussjustierungsphase beginnt nach der weiten Anlaufphase etwa 20 cm vom Ball in der x-Richtung entfernt. Ziel ist es, die Schussrichtung zu wählen (links oder rechts ins Tor) und den Nao so vor den Ball zu stellen, dass er nicht gegen den Ball läuft, aber nah genug am Ball steht, damit die ganze Schusskraft der specialAction `kick1penalty` optimal

genutzt werden kann. Die Werte für eine minimale Balldistanz zum Schuss stehen im Framework im Bereich `Infrastructure`. Natürlich müssen die Parameter feldabhängig vor Ort angepasst werden. Die Schussrichtung wird anhand zweier Möglichkeiten bestimmt und dargestellt durch eine boolsche Variable `shoot_direction`. Ist sie false, wird nach rechts ins Tor geschossen, sonst nach links. In den Symbols kann durch die Variable`goaliePos = findGoaliePos`intern in der Symbolberechnung abgeschätzt werden, wo sich ein Objekt (der Keeper) im Vergleich zur Tormitte (y=0) befindet. Wird y>=0 geschätzt befindet sich der Keeper etwas nach links versetzt und es wird nach rechts geschossen (aus der Perspektive des Strikers). Bei y<0 befindet der Keeper sich nach Schätzung etwas nach rechts versetzt und es wird dementsprechend nach links ins Tor geschossen. Einmal gewählt, ändert sich die Schussrichtung innerhalb des Versuches nicht mehr. Die Genauigkeit der Richtungsabschätzung des Keeperstandpunktes ist ausreichend für eine Nutzung, jedoch wie allgemein Gegnererkennung zur Zeit nicht optimal. Wird kein Gegner wahrgenommen, bestimmt sich die Richtung, als zweite Richtungswahlmöglichkeit, anhand eines `random`-Befehls zufällig.



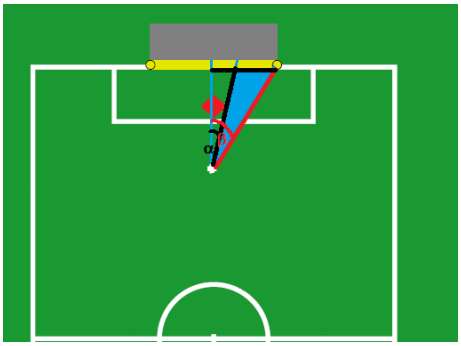
(a) Korrektur des Naos in Y-Richtung und Winkel im Verhältnis zur Tormitte

Abbildung 12.2: Tormitte erreichen

Nach der Richtungsbestimmung muss zuerst der Schütze sowohl im y-Wert mit der Tormitte übereinstimmen als auch zusätzlich eine Rotation zur Tormitte hin besitzen. Dazu wird die Rotation und der y-Wert des Strikers angepasst. Die Tormitte berechnet sich für den Stürmer aus den wahrgenommenen Torpfostenkoordinaten, die zusammenaddiert und durch zwei geteilt den Mittelpunkt des Tores ergeben. Der Schütze orientiert sich hauptsächlich relativ zum Ball. Durch den Einbezug des Torpfostens gibt es eine in absoluten Koordinaten angegebene Orientierung, die zusätzlich verhindert, dass der Stürmer zwar den optimalen Abstand zum Ball besitzt, aber beispielsweise seitlich zum Ball steht oder vor dem Ball in seine eigene Torrichtung guckend. Anfangs ist der Blickwinkel zur Tormitte natürlich schräg wie in der oberen Abbildung (vergleiche Grafik 12.2). Jedoch ist der Blickwinkel zur Tormitte nach der y-Positionsanpassung des Schützen zur Tormitte dann exakt geradeaus und somit die korrigierte Ausgangsblickrichtung rechtwinklig zur Torgrundlinie (90 Grad). In der Zeitabschnitt bevor sich der Nao jedoch genau im 90 Grad Winkel zur Torgrundlinie hinter dem Ball positioniert hat, das heißt bei einer noch abgewichenen Position, wird die Rotation zur Tormitte wie folgt berechnet (in Grafik 12.2 ist das der Winkel δ):

$$\tan\delta = \frac{Absolut(Naoposition.y - Tormitteposition.y)}{Torlinienposition.x - Naoposition.x}$$

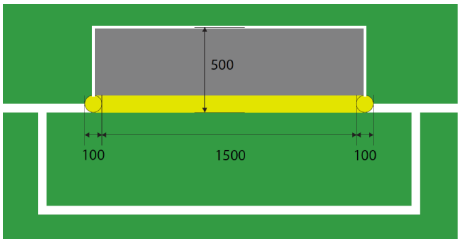
Farblich dargestellt wäre das (vergleiche Grafik 12.2) für den Zähler die orange Linie und für den Nenner die rosa Linie, da der Tangens die Gegenkathete durch Ankathete ist. Natürlich muss für den genauen Rotationswinkel zur Tormitte dann auch der Tangens gezogen werden, um einen Winkelwert zu erhalten.



(a) Nach Tormittelkorrektur des Naos, die jeweils untere Schranke α und obere Schranke β für den Schusswinkel

Abbildung 12.3: Tormitte erreichen, selbstgemachtes Bild

Nachdem sich der Schütze nochmal verbessert positioniert hat, steht er 20 cm hinter dem Ball zur Tormitte blickend. Die Schussrichtung steht zusätzlich durch das vorher beschriebene Auswahlverfahren ebenfalls fest. Für den gewünschten Schuss müssen noch zwei Positionierungen vollzogen werden. Der Nao muss sich dem Ball nähern und die momentane Rotation zur Tormitte an eine Rotation anpassen, die ihn zwischen Torpfosten und Keeper in der gewünschten Richtung schießen lässt. Zuerst muss der anzuvisierende Punkt im Tor bestimmt werden, wo der Ball im Tor landen soll. Dieser muss der Mittelwert sein aus dem Torpfosten (Maximalbereich des Schusswinkels), in dessen Richtung geschossen wird und etwa 1/3 Torlänge vom Pfosten ins Tor entfernt, damit nicht der Tormittelpunkt als minimaler Schussbereich gewählt wird. Nach Regelwerk gibt es dafür folgende Werte:



(a) Torgrößenrelationen im offiziellen Feld

Abbildung 12.4: Torgrößen, Quelle [splrules2013] S.2

Der Rotationswinkel wird dabei anhand des minimalen Rotationswinkel α und des maximalen Rotationswinkels β durch den Mittelwert abgeschätzt (vergleiche dazu Grafik 12.3 die beiden Winkel).

Der Minimalwinkel wird wie folgt berechnet:

$$\tan\alpha = \frac{Absolut(tormitte.y - goalyNotReachedPoint.y)}{tormitte.x - Naoposition.x}$$

`goalyNotReachedPoint` ist dabei ein vom Benutzer festgelegter Wert auf der Torlinie, bei dem der Keeper durch alleiniges Stehen nicht blocken kann, das heißt der Ball an ihm vorbeigeschossen wird. Je näher er an der Tormitte orientiert wird, desto größer ist natürlich der Bereich, zwischen den der Ball geschossen werden kann. Je näher der Wert zum Pfosten geht, desto enger ist der Bereich, jedoch auch unwahrscheinlicher das der Torwart den Ball hält. Bei Testreihen hat sich herausgestellt, dass etwa der Punkt bei 1/3 Torlänge vom Pfosten entfernt liegen soll. Natürlich muss auch hier der Tangens gezogen werden. Die Gegenkathete ist dabei (vergleiche Grafik 12.3) die schwarze Linie an der Torlinie von der Tormitte bis zur ersten Linienkreuzung. Die Ankathete und somit der Nenner des Bruches bildet die blaue Linie vom Nao zur Tormitte.

β ist dabei der Maximalwinkel, bei dem bei Rotationsanpassung zu seinem Winkelwert der Ball in Pfostenrichtung geschossen wird (obere Schranke). α ist der Winkel, bei dem der Ball am nächsten zum Keeper, nämlich 1/3 der Torlänge (wie oben beschrieben die Variable `goalyNotReachedPoint` vom Pfosten entfernt, geschossen wird. Da im Gegensatz zum selbstdefinierten `goalyNotReachedPoint` der Pfosten nach Regelwerk (siehe Grafik 12.4 und [splrules2013]S.2) einen festen Abstand zur Tormitte hat, kann hier ein Winkel für β direkt angegeben werden:

$$\tan\beta = \frac{Absolut(Tormitte.y - Pfosten.y)}{Tormitte.x - Naoposition.x}$$

Geht man davon aus, dass Tormitte.y=0 und Pfosten.y=Tormitte.y + $\frac{Torlaenge=1500mm}{2}$=750 mm sind, dann ergibt sich bei einem Naoabstand zur Tormitte von etwa 1800 mm (vergleiche Abstand Torlinie Elfmeterpunkt [splrules2013] S.2) zuzüglich 200 mm als Abstand Nao zum Ball nach der Justierung zur Tormitte. Dann ergibt sich als Berechnung

$$\tan\beta = \frac{750}{2000} = 0{,}375$$

und somit als Winkel für einen Maximalwert (Pfostenschuss) $\beta = 20{,}556\ Grad$. Wird der Winkel überschritten, schießt man nach der Ausrichtung der Rotation von der Tormitte auf den Winkel β am Tor vorbei.

Die gewünschte Rotationsänderung ist somit $c = \frac{\alpha+\beta}{2}$ mit dem threshold α bzw. β. Der threshold ergibt sich daraus, da nach der Mittelwertsdefinition α sowie β die Gleiche Wertdifferenz zu c haben und die untere bzw. obere Schranke für die optimale Rotation des Naos ergeben für einen Schuss zwischen Tor und Torwart. Im Programm wird der Wert des Winkels c in `TacticSymbols::getBestGoalKickAngle()` berechnet. Als letztes muss die Seitwärtsbewegung des Naos angepasst werden, indem der Ball nicht zwischen den Füßen sondern in Fußnähe des gewählten Schussbeines liegt. Dies bedeutet bei einem Schuss

nach rechts eine Verschiebung um y=-5cm, um mit dem rechten Fuß in der angegebenen Rotation zu schießen und etwa y=+5cm, um mit dem linken Fuß in der angegebenen Position nach links zu schießen. Dies wird durch ein `goto` mit gleichem threshold und x-Wert, jedoch mit einem geänderten y-Wert nach der oberen Beschreibung vollzogen. Die relative Positionierung zum Ball hilft dabei die direkten Werte `ball.y+5` oder `ball.y-5` zu übernehmen, wobei `ball.y` der Relativabstand des Naos zum Ball ist. Nun befindet sich der Nao in der gewünschten Schussposition und Rotation und muss nun schießen (dazu vergleiche Abschnitt `Der Schuss`).

12.1.4 Der Schuss

Zur Stabilisierung des Schusses hat sich herauskristallisiert, dass nach der Justierungsbewegung nicht sofort geschossen werden soll. Der Stürmer verweilt eine kurze Zeitdauer von etwa fünf Sekunden in einem Zustand (Verweildauer ist hier mit der `state_time` gleichzusetzen), wo als Ziel (mit einem `goto-` Befehl) die relative momentane Ballabstandsposition übergeben wird, damit sich der Stürmer eine Zeit lang nicht bewegt. Erst danach wird der Zustand gewechselt und der Schuss getriggert. Beim Schuss handelt es sich im Gegensatz zum Instantkick um eine specialAction. Ursprünglich handelt es sich um die specialAction `kick1hard`, bei der aus dem Stand heraus ein sehr guter Schuss möglich ist. Diese wurde nach Portierung in die specialAction `kick1penalty` überführt, da dies beim Mergen mit dem aktuellen Master verhindern soll, dass eine andere modifizierte Schussversion die ursprüngliche specialAction ersetzt und die Schusseigenschaften verändert werden. Zusätzlich wurde das Knie des Schussbeines minimal mehr angebeugt beim Herunterlassen des Beines, damit nicht in den Boden getreten wird bei manchen nicht ideal kalibrierten Naos, jedoch soweit, dass der Ball immer noch vernünftig getroffen wird. Als Parameter wird ein Boolean übergeben, der bei true die specialAction `kick1penalty` mit dem rechten Bein schießen lässt und bei false diese spiegelverkehrt für das linke Bein triggert.

12.1.5 Die Absicherung und der Beschleuniger

Bei den beiden Mechanismen `Absicherung` und `Beschleuniger` handelt es sich um alternative Zustandswege innerhalb der `Option`, die bei einer Abweichung vom Hauptelfmeterverhalten ein mögliches Fehlverhalten korrigieren sollen (Absicherung).Bei einer zu langen Ausführungsdauer, da nur eine Minute für den Schuss zur Verfügung steht, wird einen zeitlich beschleunigter Schuss ermöglicht (Beschleuniger). Das Hauptelfmeterverhalten hat einen festen Ablauf, der aber auch störanfällig sein kann. In den verschiedenen Phasen von der Initialisierung bis hin zum Schuss durchläuft das Verhalten verschiedene Zustände, in denen der Nao sich in Richtung Ball bewegt. Aufgrund der benötigten Präzision muss bei den `goto-Befehlen` ein möglichst geringer `threshold`, das heißt eine geringe Abweichung in X- und Y-Richtung und zur gewünschten Rotation, gewählt werden. Der Zustandswechsel im Hauptelfmeterverhalten ist in der Regel dadurch umgesetzt, dass innerhalb des aktiven Zustandes kontrolliert wird, ob eine vorgegebene `state_time` mindestens aufgewendet wurde und die gewünschte Position und Rotation, mit der jeweils eingestellten tolerierbaren threshold-Abweichung als obere Schranke, erreicht wurde. Offensichtlich ist es einfacher einen größeren Positions- und Rotationsthreshold anzugeben, da dadurch der Zielbereich für den `goto-Befehl` vergrößert wird und der Nao somit durch den größeren

Toleranzbereich eher innerhalb des thresholds durch Bewegungskorrekturen sich aufhält. Eine Verkleinerung des thresholds führt natürlich zu einer größeren Positionspräzision, die beim Elfmeterschießen essenziell ist, jedoch auch zu einem höheren Korrekturaufwand des Naos, da er zur Erreichung der Position weniger mögliche Freiheitsgrade besitzt und somit der `goto-Befehl` in der Regel länger ausgeführt werden muss. Im Hauptelfmeterverhalten wird ein geringer threshold vor allem bei der Justierung der Standposition vor dem Schuss gewählt. Gelangt der Nao nicht innerhalb des Toleranzbereiches durch den goto-Befehl (hierbei spielt auch der Rotationsthreshold eine Rolle), indem er ab einer gewissen Zeit nur noch leichte Korrekturen vorführt, jedoch nicht den `goto-Befehl` auf `action_done` setzen kann, wird die `Absicherung` aufgerufen. Dabei handelt es sich um einen fast identischen State, indem der Nao sich aktuell befindet, nur dass der threshold erhöht wird. Als Bedingung für den Wechsel zum alternativen Absicherungsstate muss daher in der Transition des aktuellen States eine obere Zeitschranke für die `state_time` definiert werden. Der `Beschleuniger` ist zwar einfach, aber effektiv. Jedem State wird eine gewisse Zeit zur Verfügung gestellt. Der Beschleuniger ist auch ein alternativer Statedurchlauf, wobei die Ruhephase vor dem Schuss gekürzt wird und die Rotationsthresholds etwas erhöht werden, damit der Schuss innerhalb der Minute stattfinden kann.

12.2 Torwartverhalten vom Elfmeterpunkt

Das Torwartverhalten für das Elfmeterschießen ist variabel gehalten. Für den RoboCup standen vier verschiedene Torhüter zur Verfügung. Der eine war der Standardelfmetertorwart, welcher durch Portierung ins neue Framework mit eingebunden wurde. Dieser kann sowohl blocken als auch durch Ballabschätzung mit der specialAction `dive` seitwärts springen. Der andere Torwart war der im Vorgängerkapitel beschriebene Torhüter für das RoboCup-Verhalten. Diese beiden Torhüter sind für den Gebrauch innerhalb des Torhüterstrafraums. Lässt die Regel ein Herauslaufen des Torhüters zu beziehungsweise wird dies nicht erwähnt, können die anderen beiden Torhütervarianten verwendet werden. Beim Testen des Strikers für den Elfmeterschuss stellte sich heraus, dass herauslaufende Torhüter effektiver sind als auf der Linie parierende. Dies hat mehrere Gründe, zum einen braucht der Schütze auch eine Zeitdauer, damit er den Ball erreicht, sich positioniert und schießt. Ist der Striker also zu langsam kann der Torhüter den Ball vor dem Striker erreichen, was jedoch nur gegen extrem langsame Gegner realistisch ist. Was jedoch wahrscheinlicher ist, ist dass der Keeper nah genug an den Ball gelangt, um einen größeren Torbereich hinter sich zu decken. Je näher ein Torhüter am Ball ist, desto mehr Fläche hinter ihm deckt er kegelförmig ab (vergleiche dazu Triangletaktik Kegeldeckung). Für den Schützen steht dadurch eine geringere mögliche Einschussmöglichkeit zur Verfügung. In Tests ergab ein Herauslaufen mit dem Standardelfmetertorhüter, der jedoch herauslaufen durfte eine etwa 50 prozentige Blockchance gegen den präzisen in die Torwinkel schießenden Striker. Daraufhin wurden für die Portierung direkt zwei weitere Torhüterversionen zu dem Standardelfmetertorwart und dem externen Torwart aus dem RoboCup-Verhalten entwickelt. Der erste Keeper wird `Herausläufer`, der andere `Querläufer genannt`

12.2.1 Der Herausläufer

Der Herausläufer wird nach Regelwerk in der Mitte des Tores platziert(siehe [splrules2013]S.12 f.). Wird der Gamecontroler auf `playing` gestellt, läuft der Herausläufer so lange wie möglich durch einen `goto-Befehl` mittig zum Ball. Er bleibt nur stehen, wenn der Ball sich bewegt und die `common_decision` abschätzt, dass der Ball weiter als durch Körperblocken möglich, an ihm zukünftig vorbeigeschossen wird. Daraufhin schmeisst er sich mit `dive` in die Richtung, falls diese Ballabschätzung im Seitenabstand zu ihm zutrifft.

12.2.2 Der Querläufer

Der Querläufer ist eine modifizierte Form des Herausläufers. Er geht nur eine Distanz von einem halben Meter zum Ball und hält wie der Standardelfmetertorhüter. Der halbe Meter wurde als Ausgleich zwischen Standardelfmetertorhüter und dem sehr aggressiv zum Ball gehenden `Herausläufer` gewählt. Bei ähnlichen WalkingEngines und somit Geschwindigkeiten der Naos vom Gegner und dem Keeper erreicht der Keeper den halben Meter Punkt eher als der Striker den Elfmeterpunkt, der in einem Abstand von einem Meter von ihm entfernt ist. Zwar deckt der Querläufer nicht so stark durch seinen Kegelschatten das Tor ab wie der Herausläufer, jedoch hat er auch eine gewisse Distanz, dass er noch auf Schüsse durch `dive` eher reagieren kann. Außerdem bewegt er sich leicht schräg, d.h durch einen Zufallswert geht er leicht in eine Ecke, etwa 5 cm von der Tormitte entfernt ,sodass er gerade Schüsse in die Tormitte noch abwehren kann, aber eine Torecke prädestiniert. Die prädestinierte Seite ist zufällig gewählt, kann aber auch bei Erfahrung mit Torschussecken des Gegners, wenn er etwa nur nach Rechts schießt, angepasst werden, weswegen er auch `Querläufer` genannt wird. Abschließend folgt ein kleines Fazit und Anmerkungen zur Portierung

12.3 Zwischenfazit und Portierung

Beim Elfmeterschießen handelt es sich um ein sehr komplexes Verhalten, was jedoch in der Regel nur eine Minute pro Schuss benötigt. Für einen optimalen Schuss müssen vor Ort die Parameter für den Anlauf zum Ball und die Justierung angepasst werden. Auch ein Wechsel auf einen anders kalibrierten Nao kann Probleme bereiten, sodass der Ball nicht durch die `Headcontrol` dauerhaft erkannt wird. Ist die Kamera dekalibriert, wird der Nao gegen den Ball laufen oder am Ball vorbei. Auch muss die `Headcontrol` vernünftig lauffähig sein. Ein Merge mit dem Masterbranch kann dabei Abhängigkeiten verändern, die Auswirkungen auf den Anlauf haben und somit auch die `Headcontrol`. Deshalb sollte nicht häufig gemergt werden, sondern die `cfg`-Dateien der kalibirierten Naos manuell im eigenen `Branch` angepasst werden. Die Absicherung und der Beschleuniger verbessern die Schusserfolg beim Auftritt von Zeitknappheit oder Schusspositionierungsproblemen. Die Portierung hat dabei weitreichende Konsequenzen, da der bereits fertige Schuss nicht angepasst übernommen werden kann, sondern verschiedene vorher bereits zur Verfügung stehende Options wie `Penaltybodycontrol` oder `search_for_ball` total umgeschrieben werden mussten. Auch mussten eine vernünftige Gamecontroleranbindung für die einzelnen Gamecontrolerzustände bei der Portierung nochmal seperat definiert werden, da der

Elfmeterschütze zwar vom Schiedsrichter positioniert wird, aber der Rest der Vorinitialisierung vom Gamecontroler gesteuert wird.

Teil IV

Robocup

Der RoboCup war auch dieses Jahr wieder Schauplatz für viele Teams, aus unterschiedlichen Forschungsbereichen, sich mit Anderen zu messen. Dabei geht es darum neue Entwicklungen im Bereich der Robotik zu testen und vorzustellen. Die Wettkämpfe, die eine Evaluation der vorangegangen Forschung ermöglichen, werden im Fußball ausgetragen, nicht nur wegen der Beliebtheit dieser Sportart, sondern auch wegen seines Charakters als Teamsportart, in dem es auf Kommunikation, Koordination und Geschicklichkeit des Einzelnen ankommt. Der erste RoboCup wurde 1996 in Japan veranstaltet und galt auch mit gerade einmal 8 teilnehmenden Teams als Erfolg. Dieses Jahr fand der RoboCup in Eindhoven mit 2500 Teilnehmern und mehr als 40.000 Besuchern zum 17. mal statt. Die Standard Plattform League (SPL) ist seit 2008 mit dem Aldebaran Nao auf dem RoboCup vertreten.

Kapitel 13

Challenges

Challenges sind speziell auf den Fußball bezogene Problemstellungen, die von den einzelnen Teams gelöst werden müssen. Auf dem RoboCup 2013 wurden drei unterschiedliche Challenges angeboten. Zuerst wird im folgenden Kapitel die DropIn Challenge beschrieben, in der es darum ging mit fremden Robotern aus anderen Teams zusammen zu spielen. Die Herausforderung bestand hierbei in der Koordination der Roboter untereinander. Als einziges Kommunikationsmittel war ein minimales Protokoll über WLAN zugelassen, das nicht viel mehr als die Roboter- und Ballposition beinhaltete. Die Passing Challenge, in der es um das präzise Passspiel zwischen drei Robotern geht, wird im drauf folgenden Kapitel erläutert. Zuletzt folgt die Open Challenge, in der jedes Team sich selbst eine Aufgabe stellen durfte. Dabei sind zwei Ideen für einen Open Challenge Beitrag entstanden, nämlich einen Laufbenchmark und eine Smartphone App. Die Challenges werden auf dem RoboCup den anderen Teams vorgeführt, nach festgelegten Kriterien bewertet und mit Punkten ausgezeichnet. Nach Abrechnung der Punkte aus allen drei Challenges konnte sich unser Team den dritten Platz sichern.

13.1 DropIn-Challenge

bearbeitet von: Sebastian Drywa, Christian Kroll

Die DropIn-Challenge hat die Entwicklung eines Verhaltens zum Ziel, das es ermöglicht, mit Robotern anderer SPL-Teams in einer Mannschaft zu spielen. Da jedes SPL-Team seine eigene Software implementiert, kann der Roboter nur sehr wenige Annahmen über das Verhalten seiner Mitstreiter treffen. So gibt es keine teamübergreifend definierten Spielerrollen und die nur minimal erlaubte Kommunikation sieht keine Absprachen vor. Dies macht es umso schwieriger, ein gutes Zusammenspiel zu gewährleisten und letztendlich gegen eine andere Mannschaft zu bestehen.

13.1.1 Regeln

Für diese Challenge stellt jedes teilnehmende Team entweder einen oder zwei Roboter für eine Mannschaft, welche aus insgesamt fünf Spielern besteht. Jedes Spiel dauert fünf Mi-

nuten, wobei die Spielregeln die gleichen sind wie in den normalen Spielen. Die einzige Ausnahme bildet der Torwart. Während es in normalen Spielen einen fest definierten Torwart mit Spielernummer 1 gibt, ist es in der DropIn-Challenge für jeden Roboter möglich, Torwart zu werden. Dabei gilt, wer als erster den eigenen Strafraum betritt, wird Torwart und bleibt es dann auch für den Rest des Spiels. Wenn jedoch bei der Ready-Phase keiner der Roboter in den Strafraum geht, wird bei der Set-Phase der Roboter, der am nächsten zum Strafraum steht, in diesen gestellt. Darüber hinaus besteht jede zufällig zusammengestellte Mannschaft aus Robotern von mindestens drei verschiedenen Teams. Dies gilt auch für die gegnerische Mannschaft. Dabei hängt es von der Anzahl der teilnehmenden Teams ab, wie viele Spiele gespielt werden müssen. Damit die Roboter einer Mannschaft untereinander über WLAN kommunizieren können, ist in der Challenge-Beschreibung ein simples Protokoll vorgegeben, welches im nächsten Abschnitt genauer vorgestellt und erklärt wird. Den Teams steht es dabei offen, ob sie dieses Protokoll implementieren, oder ob sie die Teamarbeit durch andere Ansätze (ohne WLAN) realisieren.

Die Bewertung der Roboter eines Teams misst sich an zwei Kriterien: Zum einen die durchschnittliche Tordifferenz der Spiele und zum anderen die durchschnittliche Bewertung einer menschlichen Jury. Diese Jury besteht aus Personen anderer Ligen, die in jedem Spiel für jeden Roboter bewerten, ob dieser gut mit den anderen Robotern im Team zusammen spielt oder nicht. Die mögliche Anzahl der erreichbaren Punkte liegt dabei jeweils zwischen 0 und 10 - 0 für einen schlechten und 10 für einen exzellenten Mitspieler.

13.1.2 Lösungsansatz

Die Implementierung der Dropin-Challenge gliedert sich in drei große aufeinander aufbauende Schritte. Eine Herausforderung bei der Umsetzung ist die Kommunikation der Spieler untereinander (im Framework-Jargon *TeamComm*). Diese kann in der bestehenden Form nicht verwendet werden, da jedes Team seine eigenen implementierungsabhängigen Protokolle nutzt. Die folgenden Abschnitte stellen die Arbeitsschritte und die Lösung für die Kommunikation der Roboter untereinander vor.

Protokoll

Wie eingangs erwähnt gibt es ein simples Protokoll, mit dem sich die Roboter des eigenen Teams untereinander verständigen können. Darunter ist zu verstehen, welche Daten an die anderen Roboter übermittelt werden und in welcher Reihenfolge und mit welchen Datentypen dies geschieht. Als Basis für dieses Protokoll sehen die Initiatoren dieser Challenge eine Struktur in Form eines `struct` der Programmiersprache `C++` vor (siehe Listing 13.1), die die Zusammensetzung der Payload der UDP-Pakete beschreibt.

Das Protokoll beginnt mit der Zeichenfolge „PkUp“, an die sich die Spielernummer und Teamfarbe anschließen. Darüber hinaus finden sich noch Werte zur Spieler- und Ballposition (inkl. deren Varianzen), zur Ballgeschwindigkeit sowie Informationen, wann ein Roboter den Ball zuletzt gesehen hat und ob er hingefallen oder *penalized* ist. Der erste Arbeitsschritt ist, dieses Protokoll ins Framework zu integrieren und dafür zu sorgen, dass es dort verarbeitet werden kann.

Listing 13.1: Originale Protokolldatei Pickup.h (mit gekürzten Kommentaren)

```
#ifndef PICKUP_H
#define PICKUP_H

static const int PICKUP_UDP_PORT = 9000; // each team will
   communicate on different ports

struct PickUpSoccerBroadcastInfo {
    char header[4];     // "PkUp"
    int playerNum;      // 1-5
    int team;           // 0 is red 1 is blue

    // position of robot
    float pos[3];       // x, y, theta
    float posVar[3];    // main diagonal of covariance matrix

    // Ball information
    float ballAge;      // seconds since last ball was seen
    float ball[2];      // position of ball
    float ballVar[2];   // main diagonal of covariance matrix
    float ballVel[2];   // velocity of the ball

    float penalized;    // seconds robot penalized or -1
    float fallen;       // seconds robot fallen or -1.
};
#endif // PICKUP_H
```

Die Integration dieses Protokolls in das Framework bedingt, aus der vorgegebenen Struktur eine komplette Representation-Klasse mit dem Namen `PickUpSoccerBroadcastInfo` zu implementieren. Da sie von der Klasse `Streamable` erbt, lässt sie sich wie jede andere Representation im Framework verwenden. Die Klasse beinhaltet neben der oben genannten Struktur und der Vererbung auch folgende optionale Variablen und Methoden, die das Framework nur intern benutzt:

- `unsigned int timestamp`
- `mutable bool isKeeper`
- `unsigned int packets`
- `unsigned int implausible`
- `bool PlausibleValues()`
- `bool isUsable() const`
- `float getDistanceToBall() const`

- `float getDistanceToRobot(PickUpSoccerBroadcastInfo const &otherRobot) const`
- `float getDistanceToPoint(Vector2<float> const & p) const`
- `bool isInMyPersonalArea(Vector2<float> const &other) const`
- `unsigned int getAge() const`

`unsigned int timestamp` speichert den Zeitpunkt des Paketeingangs. Dieser Wert dient der Berechnung des Alters der empfangenen Daten. Wenn sie zu alt sind, ignoriert das Framework die enthaltenen Werte.

`mutable bool isKeeper` gibt an, ob sich der Roboter für den Torwart hält oder nicht. Der Roboter wertet dieses Attribut grundsätzlich nur für sich selbst aus. Dieser Wert bestimmt, ob der Roboter in der `ready`-Phase zum Torwartverhalten übergeht oder nicht.

`int packets` speichert die Anzahl der empfangenen Pakete für einen bestimmten Roboter. Dies dient lediglich statistischen Auswertungen beim Debugging.

`int implausible` zählt die Anzahl der Pakete, die aufgrund von unsinnigen oder falschen Werten verworfen werden. Auch dies dient lediglich dem Debugging.

`PlausibleValues()` gibt `true` zurück, wenn zum einen der Header des Pakets korrekt ist sowie eine korrekte Team-Nummer und Player-Nummer gesendet wurden und zum anderen wenn die empfangenen Postitionswerte des Roboters nicht größer als die Feldgrenzen + 50cm sind.

`isUsable()` überprüft, ob das Paket nicht zu alt ist und ob sich der jeweilige Roboter nicht per DropIn-Protokoll als `fallen` oder `penalized` meldet. Wenn dies der Fall ist, wird `true` zurückgegeben.

Die Methoden `getDistanceToBall()`, `getDistanceToRobot(...)` und `getDistanceToPoint(...)` berechnen den Abstand des Roboters zum Ball, zu einem anderen Roboter des Teams oder zu einem bestimmten Punkt auf dem Spielfeld. Diese Methoden können benutzt werden, um z.B. zu berechnen, wer sich am nächsten zum Ball befindet.

`isInMyPersonalArea(...)` berechnet, ob ein anderer Mitspieler in einem gewissen Radius um einen selbst herum steht.

`getAge()` gibt das Alter des Paketes in Sekunden zurück. Dafür ermittelt die Methode den Zeitunterschied zwischen der aktuellen Systemzeit und der Zeit, wann das Paket eingegangen ist.

Um alle Roboter verwalten zu können, gibt es eine weitere Representation mit dem Namen `PickUpCollection`. Diese enthält vier Arrays vom Typ `PickUpSoccerBroadcastInfo`, die jeweils die Daten für jeden Roboter eines Teams aufnehmen können. Diese vier Arrays unterteilen sich in jeweils zwei für jede Teamfarbe. Es sei angemerkt, dass die Pakete der gegnerischen Mannschaft normalerweise nicht empfangbar sind, da jedes Team eigene UDP-Ports benutzt. Dies dient hauptsächlich Debugging-Zwecken und zeigt auch eventuelle Implementierungsfehler anderer Challenge-Teilnehmer auf. Zwei Arrays pro Team gibt es deshalb, weil das eine Array die letzten, und das andere die vorletzten Pakete für jeden

Roboter beinhaltet. Dies ermöglicht einfache Plausibilitätschecks der Daten und offenbart, ob bestimmte Roboter Lokalisierungsprobleme haben. Ist dies der Fall, kann das Framework das letzte empfangene Paket des betreffenden Roboters für bestimmte Berechnungen ignorieren. Dies erledigt die Methode `isPlausibleSpeed()`, die die Geschwindigkeit des Roboters von der vorletzten zur letzten empfangenen Position berechnet. Falls diese zu hoch ist, ist davon auszugehen, dass der Roboter Probleme bei der Selbstlokalisierung hat und es wird `false` zurückgegeben.

UDP-Socket

Die WLAN-Kommunikation bei der Drop-In-Challenge ist auf Socketebene lediglich eine Variation der regulären TeamComm. Daher bietet es sich an, die bereits bestehende Infrastruktur zu nutzen und an die Begebenheiten der Drop-In-Challenge anzupassen. Dazu ist allerdings ein grundlegendes Verständnis der Funktionsweise der TeamComm notwendig.

Aufbau der regulären TeamComm Die TeamComm ist im B-Human-Framework Teil des Cognition-Prozesses, der als Thread implementiert ist. Dieser kümmert sich hauptsächlich darum, in jedem Frame (einem Zeitintervall von 30 ms) Sensordaten oder Kamerabilder abzufragen und in entsprechende Daten-Queues bzw. Representations zu verpacken. Im Rahmen der TeamComm stößt er auch regelmäßig das Senden und Empfangen von Daten über das Netzwerk an.

Dazu bedient sich der Cognition-Prozess der Methode `receive` der Klasse `TeamHandler`. Der `TeamHandler` kapselt einerseits den Umgang mit dem UDP-Socket und übernimmt andererseits die Weiterleitung der Datenpakete in jeweils eine Ein- und Ausgabe-Message-Queue. Diese beiden Message-Queues stellt die Klasse selbst zur Verfügung. Sie bilden die zentrale Schnittstelle zum restlichen Framework. Einzelne Messages sind hierbei das High-Level-Äquivalent des Frameworks zur Speicherrepräsentation der UDP-Pakete, wobei sich die Typisierung allerdings nicht auf die Payload erstreckt. Lediglich Metadaten wie Größe oder Message-ID (einer ganzzahligen Konstante, um die Payload später einer passenden Representation-Klasse zuzuordnen) sind direkt aus der Message auslesbar.

Der `TeamHandler` interpretiert die Messages selbst nicht, sondern arbeitet nur mit deren Speicherrepräsentation. Er versieht sie beim Senden mit einem kleinen Header, der unter anderem einen Timestamp und eine Payloadgröße (hier inkl. Message-Container) umfasst, bevor er eine Message aus der Ausgabe-Queue so, wie sie im Speicher steht, dem Socket übergibt. Beim Empfangen kürzt der TeamHandler diesen Header wieder weg, nachdem er Größe und Timestamp geprüft hat. Auch hier landen die Daten so, wie sie vom Socket in den Puffer geschrieben werden, als Message in der Eingabe-Queue.

Der Abnehmer der eingegangenen Messages ist das Modul `TeamDataProvider`, dessen Methode `handleMessages` regelmäßig durch den Cognition-Prozess aufgerufen wird. Dies stößt die Abarbeitung aller in der Eingabe-Queue vorhandenen Messages an, wobei die Modul-Methode `handleMessage` (man beachte den Singular) jeweils für jede Message aufgerufen wird. Sie entscheidet anhand er Message-ID, in welche Representation-Instanz die Payload einer Message geschrieben wird. Erst jetzt sind die Daten vollständig typisiert und durch die anderen Module nutzbar.

Die Ausgabe-Queue wird im Rahmen der Modul-Prozesskette durch die Methode `handleTeamCommSending` (indirekt durch die Methode `execute`) des Moduls `TeamDataProvider` befüllt. Sie verpackt diverse Representation-Instanzen in Messages und platziert sie in die Ausgabe-Queue. Dies bewerkstelligt sie über das Makro `TEAM_OUTPUT` des `TeamHandler`s, das neben der Payload auch besagte Message-ID entgegennimmt. Der Cognition-Prozess wiederum stößt (nach wie vor über den `TeamHandler`) das Senden der Messages in der Ausgabe-Queue an.

Implementierung der DropIn-Challenge-Variante Die TeamComm-Komponenten werden innerhalb des Frameworks relativ häufig referenziert, da viele Module Entscheidungen anhand der Daten der anderen Mitspieler treffen. Dies hat zur Folge, dass sich die TeamComm nicht einfach für die DropIn-Challenge umschreiben lässt, weil dies zu viele Abhängigkeiten innerhalb des Frameworks in Mitleidenschaft ziehen würde. Die DropIn-Kommunikation muss daher parallel zur TeamComm laufen. Damit letztere trotzdem nicht in den Äther vordringt, ist es notwendig, die `send-` und `receive`-Methoden des `TeamHandler`s so zu ändern, dass weder Daten in das UDP-Socket gelangen, noch Daten davon in der Eingabe-Queue landen. Es genügt, die Codezeilen, die das Socket-Handling betreffen, auszukommentieren. Auf diese Weise bleibt die TeamComm an sich intakt, wenn auch mit einem simulierten Paketverlust von 100%.

Der nächste Schritt zur DropIn-Kommunikation ist die Implementierung einer eigenen `TeamHandler`-artigen Klasse, die in diesem Fall einfach `DropInChallenge` heißt. Auch hier gibt es wieder eine Ein- und Ausgabe-Queue, die aber nur eine einzige Art von Message aufnimmt, die die oben vorgestellte `PickUpSoccerBroadcastInfo`-Representation umfasst. Das Grundgerüst ist weitestgehend identisch, lediglich die `send-` und `receive`-Methoden sind angepasst.

Da es sich bei den UDP-Paketen nicht mehr um die binäre Speicherrepräsentation einer Message handelt, muss die `DropInChallenge`-Klasse diese zunächst aufbereiten. Die neu hinzugekommene Methode `decipherPacketBuffer` nimmt die Speicheradresse des vom Socket gefüllten Puffers entgegen und liest die einzelnen Werte entsprechend der Offsets des `C++-struct`s aus. Die so erhaltenen Werte speichert sie in einer `PickUpSoccerBroadcastInfo`-Representation, wobei sie gleichzeitig alle Koordinaten gemäß Framework-Standard von Zentimetern in Millimetern umrechnet (ebenso die Geschwindigkeiten von cm/s in mm/s). Die Methode trägt außerdem noch die aktuelle Systemzeit als Timestamp ein und führt erste Plausibilitätstests durch, um die Zähler für alle Pakete insgesamt bzw. die unbrauchbaren Pakete (für die Debug-Statistik) zu aktualisieren. Die übergeordnete `receive`-Methode fügt die Representation im Falle fehlgeschlagener Tests nicht in die Message-Queue ein, so dass das Framework die Daten gar nicht erst weiter verarbeitet.

Das Gegenstück zu `decipherPacketBuffer` ist die Methode `fillPacketBuffer`, welche die `PickUpSoccerBroadcastInfo`-Representation bytegenau im Format des `C++-struct`s in einen Puffer serialisiert. Ebenso konvertiert sie die Werte in die durch die Challenge vorgegebenen Einheiten zurück. Durch diesen Ansatz fallen andauernde und fehleranfällige Konvertierungen in den Modulen des Frameworks weg.

Die DropIn-Kommunikation benötigt auch ein eigenes, angepasstes Modul, das der Klasse `TeamDataProvider` der TeamComm entspricht, der sogenannte `PickUpSoccerBroadcast-`

`Provider`. Dies ist eine vereinfachte Form des `TeamDataProvider`s, die lediglich mit den spärlichen Informationen des Challenge-Protokolls umgehen muss. Auch hier gibt es wieder eine Methode `handleMessages`, die nichts weiter tut, als die Representation `PickUpCollection` mit den Daten aus der Eingabe-Queue zu füllen und die Attribute für die letzten und vorletzten Werte zu aktualisieren. Die Methode `execute` ist geringfügig komplexer, da sie die Werte anderer Representations (u.a. Welt-Modell) miteinbezieht, um die eigene zu sendende `PickUpSoccerBroadcastInfo`-Representation mit sinnvollen Werte zu füllen.

Zu guter Letzt bedarf der Cognition-Prozess an all den Stellen einer Ergänzung, wo er den `TeamHandler` bzw. das Modul `TeamDataProvider` der TeamComm anspricht, so dass er auch die entsprechenden Gegenstücke der DropIn-Implementierung einbindet.

DropIn-Verhalten

Der letzte Arbeitsschritt ist das Verhalten. Es ist ein simples Verhalten, welches nur drei Fälle abfragt und dementsprechend handelt:

- Wenn der Roboter am nächsten zum Ball steht, soll er sich zum Ball bewegen
- Wenn ein Mitspieler am nächsten zum Ball steht und der abfragende Roboter selber in der eigenen Hälfte steht, soll er eine Position zwischen Ball und Tor einnehmen und verteidigen
- Wenn ein Mitspieler am nächsten zum Ball steht und der abfragende Roboter selber in der gegnerischen Hälfte steht, soll er eine unterstützende Position einnehmen, um bei einem Pass nach vorne weiterspielen zu können

Das Grundgerüst für dieses Verhalten ist das gleiche wie für das Verhalten, welches beim Robocup 2013 zum Einsatz kommt. Dies erlaubt ohne großen Aufwand die Übernahme von Dingen wie Button-Interface, LED- und Sound-Einstellungen. Der State `ready` und der State `playing` bedürfen jedoch für das Verhalten weiterer Anpassungen. Da die reguläre TeamComm nicht zur Verfügung steht, ist es leider nicht möglich, die bereits im Framework integrierten Symbole zu benutzen, da die meisten davon auf die TeamComm angewiesen sind, die nie sinnvolle Pakete erhält.

Die benötigten Berechnungen nimmt deshalb die neue Symbol-Klasse `DropInSymbols` vor, die nur Daten des eigenen Roboters und die empfangenen Daten des DropIn-Challenge-Protkolls verwendet. Dort gibt es die Variablen `OptPosition`, `OptAngle`, `isKeeperNow` und `nearestToBall`, die die Methoden `void calcOptKickOffPosition()` (für den State `ready`) und `void calcOptPosition()` (State `playing`) berechnen bzw. setzen.

`void calcOptPosition()` berechnet zuerst, ob der Roboter am nächsten zum Ball ist. Falls dies der Fall ist, wird `nearestToBall` auf `true` gesetzt. Falls dies nicht der Fall ist, berechnet die Methode die optimale Position auf dem Feld, wie es in den oben beschriebenen Fällen angegeben ist und aktualisiert die Variablen `OptPosition` und `OptAngle`, um diese anzusteuern.

`void calcOptKickOffPosition()` bestimmt hingegen die optimale Kick-Off Position. Die Methode schaut dabei mehrere festgelegte Zielpunkte an und berechnet, welchem dieser Punkte der Roboter am nächsten ist. Diesen Punkt speichert sie dann wieder in `OptPosition` und `OptAngle`. Dies gilt auch für den Torwartpunkt.

Da im DropIn-Verhalten jeder Roboter Torwart werden kann, ermittelt die Methode `calcOptKickOffPosition()` zusätzlich, ob die Positionen anderer Roboter im eigenen Strafraum liegen. Falls dies der Fall ist, setzt sie die Variable `isKeeper` von `PickUpSoccerBroadcastInfo` dieses Roboters auf `false` und dadurch wird der Torwart-Zielpunkt nicht mehr berücksichtigt. Wenn noch keiner der Roboter Torwart ist und der Roboter selber in den Strafraum geht, setzt die Methode den Wert der Variablen des eigenen Roboters auf `true` und sieht nur noch den Torwart-Zielpunkt als optimalen Punkt an.

Wenn der State von `set` auf `playing` wechselt, fragt das Verhalten ab, ob der Roboter selbst Torwart ist. Ergibt diese Abfrage den Wert `true`, geht es zum Torwartverhalten über, ansonsten zum normalen Spielerverhalten. Für den Torwart kommt auch hier das Torwartverhalten zum Einsatz, welches Gegenstand eines vorherigen Kapitels ist.

13.1.3 Wettkampf

Insgesamt haben an der DropIn-Challenge sechs Teams teilgenommen und es wurden vier Spiele ausgetragen. In jedem Spiel haben die Nao Devils ein oder zwei Roboter gestellt. Die Implementierung des Protokolls hat einwandfrei funktioniert, was die Überprüfung der Arrays nach dem letzten Spiel gezeigt hat. Die Roboter von B-Human und von den Nao Devils haben rege Daten ausgetauscht. Jedoch hat es auch Schwierigkeiten von anderen Teams gegeben. Manche Teams haben erst gar kein Protokoll implementiert, während andere Teams zwar Daten geliefert haben, jedoch zum größten Teil nichts verwertbares (z.B. ungültige Fließkommazahlen als Positionsdaten). Ein weiteres Team hat das Protokoll mit prinzipiell vernünftigen Werten gesendet, nur leider mit der falschen Teamnummer, wodurch das Framework die Daten ignoriert hat.

Das implementierte Verhalten hat sich als sehr stabil herausgestellt. So hat ein Roboter der Nao Devils als Torwart mit einer Parade glänzen können. Ein weiterer hat ein Tor geschossen. Dies hat dem Team zum Schluss einen guten zweiten Platz in dieser Challenge und damit 24 Punkte für die Gesamtbewertung eingebracht.

13.2 Passing-Challenge

bearbeitet von: Jan-Hendrik Berlin, Yuri Struszczynski

Um eine gute Platzierung in der Teamwertung zu erreichen, ist es wichtig an allen angebotenen Challenges teilzunehmen. Somit war die Teilnahme an der Passing-Challenge-2013 ein Herausforderung der wir uns gerne gestellt haben.

13.2.1 Spielregeln

In der Passing-Challange sollen sich 3 Roboter den Ball im Kreis zu passen. Die Roboter werden in einem von den Veranstaltern definiertem Dreieck auf dem Spielfeld positioniert. Um jeden Roboter ist ein Kreis mit einem Radius von 35cm markiert. Der Ball muss aus diesem Kreis in den Kreis eines anderen Roboters gespielt werden. Dabei soll vermieden werden den Ball zu dem Roboter zurück zu spielen, von dem man den Ball bekommen hat. Dies wird durch eine klar definierte Punktevergabe gezielt gefördert. Die Passing-Challenge dauert 3 min pro Team. In dieser Zeit müssen möglichst viele Pässe gelingen. Die Koordinaten von den Robotern werden einen halben Tag vor der Passing-Challenge bekannt gegeben. j

13.2.2 Lösungsansatz

Allgemeiner Ansatz

Es stellte sich schnell heraus, dass es drei wesentliche Punkte gibt, an denen gearbeitet werden muss.

- Kick
- Verhalten
- Lokalisierung am Start

Für den Kick wurde eine bereits vorhandene KickEngine benutzt. Mithilfe der KickEngine kann man einen Kick auslösen, der den Ball an ein festgelegtes Ziel schießt. In Versuchen zeigt sich die KickEngine als bedingt brauchbar. Mangels besserer Alternativen wird jedoch trotzdem auf diese KickEngine zurück gegriffen.

Das Verhalten soll möglichst einfach gestrickt sein und auf zusätzlichen Datenaustausch via WLAN verzichten. Es wird festgelegt, dass der Ball immer gegen den Uhrzeigersinn gespielt wird. Dies hat den Vorteil, dass man Anhand der Ballposition entscheiden kann was ein Roboter tut soll. Kommt der Ball schnell auf den Roboter zu, soll dieser Ihn anhand einer SpecialAction annehmen. Ist ein Roboter am nächsten zum Ball, wird dieser den nächsten Pass ausführen.

Die Lokalisierung muss nur für den Start angepasst werden. Sobald der GameState auf „playing“wechselt, wird im Regelfall eine Hypothese gespornt, in welcher der Roboter auf der eigenen Hälfte im Seitenaus des Spielfeldes steht. In der vorgestellten Challenge sind die Positionen jedoch fest definierte Koordinaten auf dem Spielfeld. Die Roboter sollen also beim Wechsel auf den GameState „playing“nicht einlaufen, sondern sich nur kurz lokalisieren und dann die entsprechende Position halten. Hier zu sind minimale Anpassungen an der Selbstlokalisierung der NAOs erforderlich, welche im folgenden Abschnitt beschrieben werden.

Lokalisierung

Im SelfLocator2012 (SelfLocator2012.cpp), welcher Teil des WorldModelGenerators ist, gibt es bereits eine Funktion die die Hypothesen für einen normalen Spielbegin spornt. Auf Basis dieser Funktion (addHypothesesOnInitialKickoffPositions) wurde dann eine neue Funktion eingefügt, die entsprechend Änderungen an den Starthyphothesen der Roboter vornimmt. Damit der NAO sich besser für die neuen Hypothesen entscheiden kann, wurden die Hypothesen der InitialKickoffPosition entfernt.

Die Positionen die die Hypothese definieren, sollten dabei zuerst aus der positionsByRules Konfigurationsdatei geladen werden. Da jedoch für die Challenge nur 6 Werte bzw. Koordinaten definiert werden müssen, sind die Hypothesen nun hart im Code definiert. Dies hat außerdem den Vorteil, dass die Roboter schneller zueinander ausgerichtet werden können, da die NAOs mit Blickrichtung in das Dreieck aufgestellt werden und diese ebenfalls in Rad einkodiert werden muss.

Verhalten

Das Verhalten basiert auf dem Grundgerüst des Turnierverhalten, welches zum Robocup 2013 eingesetzt wird. Die Roboter sollen die Spielernummern 2,3 und 4 haben. Berechnungen werden in der neu angelegten Symbols-Klasse „Passing_Symbols.h “vorgenommen. Folgende Funktionen wurde hierzu implementiert:

- Vector2_D<double> PositionByNumber(int robotnr)
- Pose2D_D ownPosition()
- Pose2D_D Target()
- Pose2D_D Source()
- Pose2D_D calcOptKickPosition(Vector2_D<double> optKickTarget)
- bool getKickToOwnPosition()
- bool getGoToBall()

Vector2_D<double> PositionByNumber(int robotnr) gibt einen Vektor mit Feldkoordinaten abhängig von der übergebenen Spielernummer zurück.

Pose2D_D ownPosition() gibt die eigene Ausgangsposition mit einem Winkel zurück. Diese Informationen kann man sehr gut im Datentyp „Pose2D_D“speichern. Die Ausgangsposition wird über einen Aufruf von „PositionByNumber(int robotnr)“ermittelt, wobei hier „int robotnr“die eigene Spielernummer ist. Der Winkel zeigt in Feldkoordinaten in Richtung der Position, von der aus der Ball erwartet wird.

Pose2D_D Target() und **Pose2D_D Source()** sind sehr ähnliche Funktionen. Pose2D_D Source() gibt an woher der Ball erwartet wird und Pose2D_D Target() gibt an

Listing 13.2: die Funktion Vector2_D<double> PositionByNumber(int robotnr)

```
// initial points on the field given by tecnical comitee
Vector2_D<double> PassingSymbols::PositionByNumber(int
   robotnr)
{
        Vector2_D<double> pos(0,0);

        switch(robotnr)
        {
         case 1: return  PositionByNumber(4); // target of
            number 2
         case 2: return
            Vector2_D<double>(paramPos.r2x,paramPos.r2y); //
            position robot 2
         case 3: return
            Vector2_D<double>(paramPos.r3x,paramPos.r3y); //
            position robot 3
         case 4: return
            Vector2_D<double>(paramPos.r4x,paramPos.r4y); //
            position robot 4
         case 5: return  PositionByNumber(2); // source of
            number 4
         default: return Vector2_D<double>(0,0); //default
            should never match, but it must exist...
        }
}
```

wo der Ball hin gespielt werden soll. Die Koordinaten der Funktionen werden wieder über einen Aufruf von „PositionByNumber(int robotnr)“ermittelt, wobei hier „int robotnr“**nicht** die eigene Spielernummer ist. Für Pose2D_D Target() wird die eigene Spielernummer-1 übergeben. Für Pose2D_D Source() wird die eigene Spielernummer+1 übergeben. „PositionByNumber(int robotnr)“gibt dann die jeweils richtigen Koordinaten aus. (siehe hierzu „case 1“und „case 2“im Listing „PositionByNumber“. Die Winkel zeigen auf das Ziel bzw. auf die Quelle des Balls.

Pose2D_D calcOptKickPosition(Vector2_D<double> optKickTarget) errechnet eine möglichst optimale Position, um den Ball zu schießen. Diese Funktion wurde aus dem Tunierverhalten kopiert und leicht angepasst. Im Tunierverhalten ist als Ziel immer das Tor angegeben, für die Passing-Challange wird das Ziel übergeben.

bool getKickToOwnPosition() dient zur Entscheidung, ob der Ball in den eigenen Kreis zurück gelegt werden soll. Da man nur Punkte erzielen kann, wenn der Ball aus dem Kreis des schießenden Roboters geschossen wird, ist es in bestimmten Situationen sinnvoll den Ball erst wieder in den eigenen Kreis zu legen. Tests haben ergeben, dass es sinnvoll ist, den Ball zurück zulegen, falls er mehr als 1m vom Mittelpunkt des Kreises entfernt liegt.

bool getGoToBall() entscheidet ob der Roboter zum Ball geht. Die Entscheidung wird Anhand der Entfernung des Balls zu den einzelnen Kreisen gefällt. Ist der Ball als nächstem zu einem Kreis, wird der Roboter, der in dem Kreis steht zu dem Ball gehen. Falls der Ball in der Mitte zwischen 2 Kreisen liegt (50% der Strecke), könnte es ein Deadlock geben. Um dies zu vermeiden geht ein Roboter zum Ball, wenn er bis zu 60% der Strecke zwischen den beiden Kreisen entfernt liegt.

Änderungen werden in einer cabsl-Option vorgenommen, welche im GameStae „playing "aktiv ist. Diese Option heißt „Playing_Passing.h "und hat folgende States:

- positioning (initialer State)
- kickToOwn
- kick
- takeball
- localize

Es gibt eine „common_transition", die den Fall abfängt, dass der Roboter delokalisiert ist. In diesem Fall wird sofort der State „localize"aufgerufen. Der State **„localize"** lässt den Roboter eine Kopfbewegung ausführen mit welcher der Roboter sich leichter wieder lokalisieren kann. Nach 2 Sekunden gibt es einen Übergang zum State „positioning ".

Der State „**positioning** "wird initial aufgerufen. Der Roboter positioniert sich in der Mitte seines eigenen Kreises und blickt in zu dem Mitspieler, von dem er den Ball erhalten wird. Als action wird die cabsl-Option „standInPosition"gestartet. Diese Option ist auch für die Passing-Challange geschrieben. Sie sorgt dafür, dass der Roboter seine Position ein nimmt und die „special_action stand"ausführt. Dank der „special_action stand"kann der Roboter schneller den Ball blocken, wozu er in der Option „Playing_Passing.h "in den State „takeball"springen würde.

Der State „**takeball**"dient dazu den Ball zu blocken, wie es der Keeper des Tunierverhaltens auch macht. Es wird die „special_action keeperBlockLeft"ausgeführt. Danach gibt es einen Übergang zum State „positioning".

Die States **„kickToOwn"** und **„kick"** werden von dem State „positioning"aus erreicht. Falls der Ball erst in den eigenen Kreis gelegt wird (siehe oben „getKickToOwnPosition") gibt es einen Übergang zum State„kickToOwn". Soll der Ball direkt zum anderen Roboter gespielt werden, wird der State „kick"aktiviert. In Beiden States wird ein Kick ausgelöst, nur das Ziel unterscheidet sich. Einmal wird der eigene Kreis angespielt, das andere mal das eigentliche Ziel.

13.2.3 Wettkampf

Das Verhalten stellte sich während der Passing-Challange als stabil heraus. Die zu Beginn gespanten Hypothesen schienen richtig und genrell gab es damit keine offensichtlichen

Probleme mit der Lokalisierung der Roboter. Die Gelenke der Roboter waren jedoch zum Zeitpunk der Passing-Challange ziemlich heiß, was auf die vorher stattgefundenen Spiele und das Tuning für die eigentliche Challenge zurückzuführen ist. Aus diesem Grund, waren die durchgeführten Pässe ungenau, wobei die Richtung und der Ansatz jedes Mal stimmte. Defakto blieben die Bälle jedoch bei jedem Schuss, wenige Zentimenter vor oder neben dem Kreis liegen, womit kein Punkt erzielt wurde. Aus diesem Grund sah das Ergebnis im Vergleich zur Konkurrenz zwar nicht schlecht aus, erzielte jedoch keine Punkte, was zu einer Platzierung bei dieser Challenge führte, die im direkten Vergleich mit den restlichen Teams unbefriedigend war.

13.3 Open-Challenge

Die Open-Challenge des RoboCups 2013 ist ein Wettbewerb mit frei wählbaren Inhalt. Jedes Team darf aktuelle Forschungsergebnisse präsentieren, solange sie zum Themengebiet der SPL passen. Debug-Tools sind dabei explizit ausgeschlossen. Nach einer Minute, die für den Aufbau reserviert ist, präsentiert jeder Teilnehmer seinen Beitrag in einem drei minütigen Vortrag. Bei der Open-Challenge wurde mit der Android-App SmartRef angetreten. Die App wird nach der Beschreibung eines Laufbenchmarks vorgestellt. Der Benchmark war als ursprünglicher Beitrag geplant, wurde dann aber aus mehreren Gründen eingestellt.

13.3.1 Entwurf eines Laufbenchmarks

bearbeitet von: Florian Gürster

Ein Kernforschungsbereich des Instituts für Roboterforschung ist der Lauf der humanoiden Roboter. Obwohl es mehrere verschiedene Läufe gibt, existiert noch keine Grundlage mit der man diese Läufe wissenschaftlich vergleichen kann. Die Idee hinter dem Entwurf eines Benchmarks war verschiedene Kriterien zu definieren, welche einen guten Lauf ausmachen, und eine faire Vergleichsbasis zu schaffen. Ein Risiko bei der Akzeptanz eines Benchmarks ist, dass bei einer Teilnahme die eigene Arbeit als schlecht dastehen kann und deshalb keine Teilnahme riskiert wird. Die Hemmschwelle sollte gesenkt werden, indem der Test aus mehreren Disziplinen besteht, in denen jeder Teilnehmer gewisse Stärken einfließen lassen kann. Ein guter Lauf zeichnet sich durch eine hohe Geschwindigkeit, eine hohe Genauigkeit in der Ausführung und die geringe Anfälligkeit gegenüber nicht idealen Umgebungsgegebenheiten aus.

Um möglichst alle Laufeigenschaften gleichwertig zu erfassen, wurde ein Parcour mit mehreren Stationen entworfen (siehe Abb. 13.1). Durch die Möglichkeit zwei Teams gegeneinander auf einem Spielfeld antreten zu lassen, soll die Wertung auch für die Zuschauer interessant werden.

Jede Station hat eine eigene Teilwertung. Bei jeder Station soll die Zeit gemessen werden, die für das Absolvieren benötigt wird. Die Messung der Zeit kann mit Lichtschranken durchgeführt werden. Damit zu verhindern, dass bei dem Scheitern in einer Disziplin die Gesamtwertung zu stark beeinflusst wird, muss es für jede Station ein Zeitlimit geben. Von diesem Zeitlimit wird die Zeit, die für die Durchführung benötigt wurde, abgezogen. Sollte

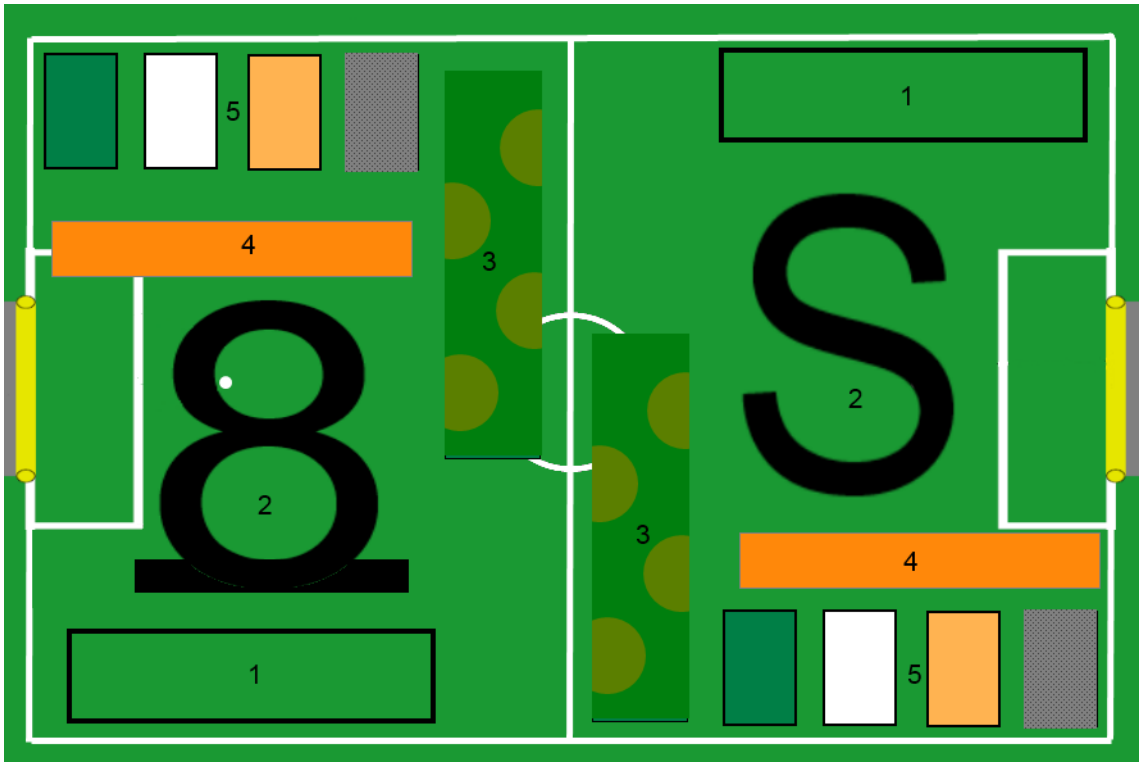


Abbildung 13.1: Nicht proportionale Skizze des Parcours für den Laufbenchmark mit Stationsnummern

es stationsspezifische Parameter geben fließen sie gesondert ein. Die Gesamtwertung ergibt sich über die Summe aller Zeiten. Die erste Station ist ein klassischer Sprint, der Roboter muss eine gewisse Strecke möglichst schnell zurück legen. Neben der Zeit gibt es keine anderen Einflussgrößen. Bei der zweiten Station wird die Präzision gemessen. Der Teilnehmer muss möglichst genau einer vordefinierten Strecke folgen. Jede Abweichung von dem vordefinierten Pfad gibt eine Strafzeit. Die Abweichung sollte durch eine Deckenkamera oder ein ähnliches System festgestellt werden. Die dritte Station soll die Stabilität auf Unebenheiten überprüfen. Unter dem Teppich sollten in vordefinierten Abständen Erhebungen angebracht werden, um eine unebene Fläche zu erhalten. Der Umgang mit einen federnden Untergrund wird bei der vierten Station geprüft Der Roboter geht über ein Brett, dass durch den Roboter in Schwingung gebracht wird. Die fünfte und letzte Station erforscht den Umgang mit unterschiedlicher Griffigkeit des Bodenbelags. Jede Station müsste mit standardisierten Bodenbelegen und Untergrundstrukturen ausgestattet werden, damit eine Vergleichbarkeit über längere Zeit erhalten bleibt.

Die Idee des Laufbenchmarks als Beitrag zur Open-Challenge wurde aus mehreren Gründen nach dem Entwurf eingestellt. Für jede Station hätte ein Beispiel gebaut werden müssen und auch innerhalb einer Minute aufgebaut werden können. Jede Station hätte mit aufwändigen Messungen in verschiedene Normen spezifiziert werden müssen. Die Präsentationszeit von drei Minuten hätte nicht für eine Vorführung oder ausführliche Beschreibung gereicht. Eine Reduzierung der Stationen hätte den Charakter des Benchmarks stark verfälscht und nicht mehr zu den ursprünglichen Vorsätzen gepasst. Als Realisationszeitraum war etwas mehr als ein Monat vorgesehen. Das ausbalancieren des Tests, hätte viele Tests benötigt.

Viele der Roboter hatten verschiedene mechanische Defekte. Die Roboter, die zur Verfügung standen, sollten auf Anweisung der Teamleitung für das Turnier geschont werden. Andere Projektgruppenteilnehmer hatten schon Probleme, für ihre Tests eine Auswahl zusammenzustellen, um aussagekräftige Ergebnisse zu erhalten. Bei der Optimierung auf die einzelnen Stationen, waren weitere Schäden zu erwarten und diese hätte zu erheblichen Problemen bei fast allen anderen Aufgaben geführt.

Um die Roboter zu schonen, wurde eine Alternative gesucht, welche ohne Roboter auskommt. Als Lösung ist eine Smartphone-Anwendung entstanden, die die Schiedsrichter im Spiel unterstützen kann. Diese App mit dem Namen SmartRef wird im folgenden Abschnitt beschrieben.

13.3.2 SmartRef

bearbeitet von: Bastian Böhm, Florian Gürster

Nachdem sich die ursprüngliche Idee eines Laufbenchmarks (siehe Kapitel 13.3.1) als unpraktikabel für eine Fertigstellung bis zum RoboCup 2013 gezeigt hatte, musste eine Alternative gefunden werden. Um im Themenbereich der SPL zu bleiben, wurde sich für die Unterstützung der Schiedsrichter entschieden. Ein SPL-Spiel wird von mindestens vier Schiedsrichtern geleitet. Das Schiedsrichterteam besteht aus dem Spielleiter, dem Hauptschiedsrichter und zwei Assistenzschiedsrichtern. Der Spielleiter gibt die Anweisungen des Hauptschiedsrichters an einem Computer in die GameController-Anwendung (Abb. 13.2) ein, damit diese über WLAN an die spielenden Roboter gelangen. Die Assistenzschiedsrichter führen die Anweisungen des Hauptschiedsrichters aus. Zu den Aufgaben der Assistenten gehören zum Beispiel das Entfernen und Positionieren der Roboter und das Zurücklegen des Balles an bestimmte Positionen, nachdem er ins Aus gerollt ist. Haupt- und Assistentsschiedsrichter halten sich am Rand des 54 m^2 großen Spielfeldes auf, um das Geschehen gut zu überblicken und schnell eingreifen zu können. Bei der Teilnahme an den GermanOpen 2013 hat sich gezeigt, dass ein schnelles und störungsfreies Eingreifen der Schiedsrichter durch mehrere Faktoren behindert wurde.

Die Abstände und die Umgebungsgeräusche, hervorgerufen von anfeuernden Teams und Publikum, behindern den Informationsfluss der Schiedsrichter und Unsicherheiten im Regelwerk sorgen für Verzögerungen und sogar falsche Umsetzung der Regeln. Als Lösung wurde eine Smartphone-Anwendung mit dem Namen SmartRef angestrebt. SmartRef nutzt die bestehende Infrastruktur und benötigt keine Modifikationen an der GameController-Anwendung und der Robotersoftware. In den ersten Überlegungen sollten die Schiedsrichter durch die App die Möglichkeiten des GameControllers erhalten. Das aktive Eingreifen der Schiedsrichter in den Funkverkehr hat sich sehr früh in der Konzeptionierungsphase als falsch erwiesen. Durch die Möglichkeit selber zu senden, wäre die Ablenkung der Schiedsrichter erhöht worden und die Roboter hätten widersprüchliche Anweisungen erhalten können. In der realisierten Form hat die Anwendung zum größten Teil die Aufgabe der Visualisierung des Funkverkehrs zwischen GameController und Robotern.

Als Zielsystem wurde das Betriebssystem Android ausgewählt. Android hatte im Juli 2013 einen Marktanteil von 70,4% in den bevölkerungsreichsten europäischen Ländern im Be-

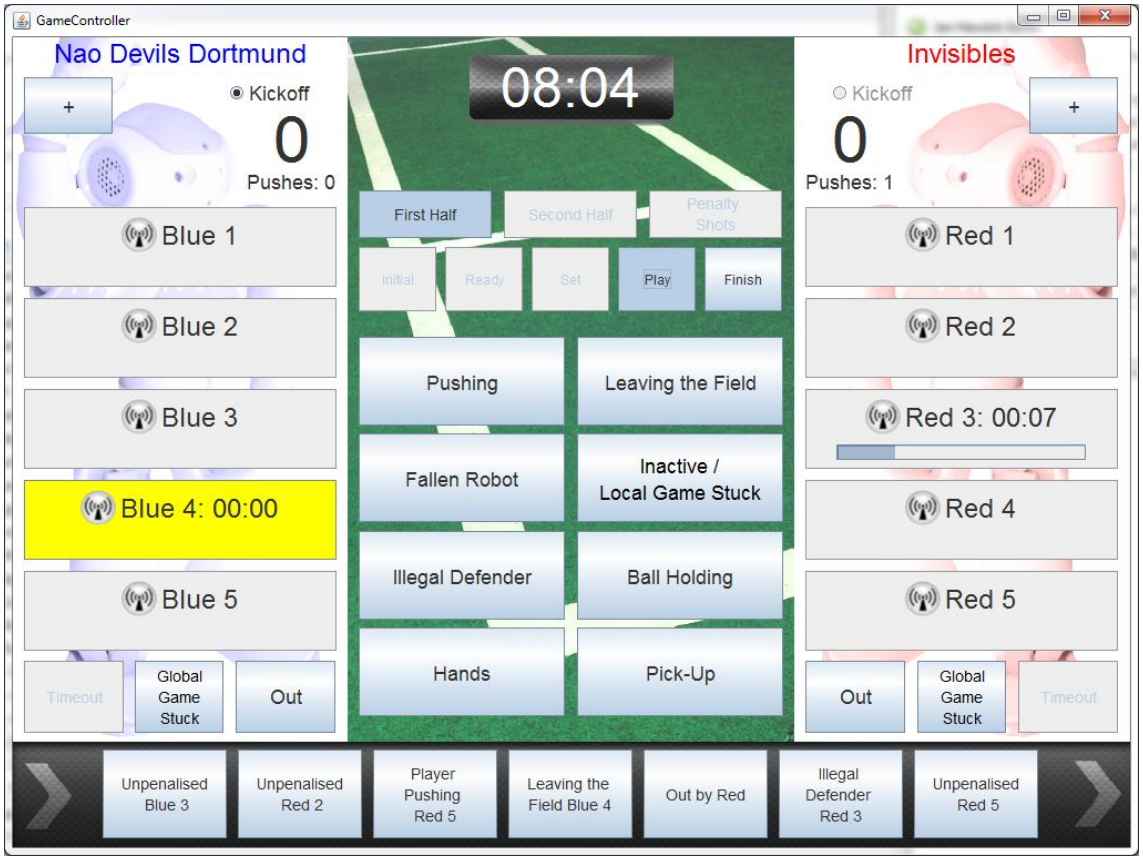


Abbildung 13.2: Oberfläche der GameController-Anwendung

reich der mobilen Geräte[1], weshalb ein großer Nutzerkreis erreicht werden kann. Einstieg in die Entwicklung ist durch frei zugängliche Entwicklungsumgebungen äußerst einfach[2]. Die Entwicklung der Android-Anwendungen geschieht zum größten Teil in Java, es ist aber auch möglich, einzelne Programmteile in C und C++ zu realisieren. Durch die weite Verbreitung des Systems konnte auf das vorhandene und passende Framework AndEngine zurückgegriffen werden.

AndEngine

Die AndEngine[3] ist ein Open-Source 2D Framework für Android und verwendet OpenGL für die graphische Darstellung.

Einer der Gründe für die Verwendung der AndEngine für SmartRef war, dass die App viele Grafiken und Gui Elemente verwalten muss. Zusätzlich ist es, für solch eine App mit einer sich dynamisch ändernden Oberfläche, notwendig die Texte und Grafiken an die gewünschten Stellen zu setzen und für den Nutzer sichtbare Elemente kontinuierlich neu zu positionieren. Hier bietet sich die AndEngine besonders an, da das Kernelement des

[1] [20] Beiersmann, Stefan: *Android erreicht 70% Marktanteil in Europa.* http://www.zdnet.de/88160639/android-erreicht-70-prozent-marktanteil-in-europa/, 2013

[2] [21] Google: *Android developers.* http://developer.android.com/reference/, Juni 2013

[3] [22] Gramlich, Nicolas: *Andengine.* https://github.com/nicolasgramlich/AndEngine/, 2013

Frameworks auf der Spieleentwicklung liegt. Viele Elemente der Spieleentwicklung, wie unter anderem die Verwendung von Grafiken, verschiedene Level und Animationen, treffen auch auf die SmartRef App zu. So lag es nahe, für die Entwicklung eine Spieleengine zu verwenden.

Empfangen und Auslesen der Pakete

Die Grundlage für SmartRef ist das reibungslose Empfangen von Paketen, die vom GameController gesendet werden. Hierzu wurden mehrere Klassen entworfen, die für das Empfangen und Umwandeln der Datenpakete in Java Strukturen zuständig sind. Daten werden in Form von UDP Paketen an die Roboter auf dem Feld übermittelt. Den Inhalt der UDP Pakete findet man in Anhang XY. Bei der Implementierung wurde als Grundlage der schon vorhandene Code von B-Human benutzt, welcher auch im GameController[4] Verwendung findet, um die Pakete in Datenstrukturen zu überführen. Da die Android App ununterbrochen auf neue Pakete warten soll, ist es notwendig das Empfangen und Verarbeiten von Paketen in einer Endlosschleife zu erledigen. Um ein Blockieren der GUI zu verhindern während sie auf Pakete vom GameController wartet wird die in Android vorhandene Klasse `AsynchTask` verwendet.

Der Vorteil der `AsynchTask` Klasse ist, dass die Klasse sich um die Verwaltung des Threads kümmert und Methoden bereitstellt, um den Fortschritt und die Fertigstellung des Threads an andere Klassen beziehungsweise Methoden weiter zu geben. Eine eigene Klasse muss `AsynchTask<Params, Progress, Result>` erweitern und dabei drei Parameter spezifizieren. `Params` steht dabei für Parameter die der `doInBackground` Methode beim Starten des asynchronen Tasks übergeben werden. Die `doInBackground` Methode wird in der Implementierung überschrieben und führt Code des asynchronen Tasks im Hintergrund aus. Der `Progress` Parameter beschreibt den Datentyp, der für die Methode `onProgressUpdate` verwendet wird. Diese Methode wird für Fortschrittsaktualisierungen genutzt. Der letzte Parameter ist `Result` und beschreibt den Datentyp für die Methode `onPostExecute`. Aus dem Grund, dass der asynchrone Task niemals terminiert, benötigt man `onPostExecute` nicht. Die Methode kann man aber dazu verwenden, um zum Beispiel die Gui über einen abgeschlossenen Ladevorgang zu informieren.

Listing 13.3: Ausschnitt aus NetworkReceive.java

```
public class NetworkReceive extends AsyncTask<Object,
   DatagramPacket, String> {

        @Override
        protected String doInBackground(Object... params) {
                /*
                * Neuen Socket auf Port 3838 oeffnen
                */
                while(true) {
                /*
```

[4] [23] Bartsch, M., M. Steinbeck, R. Wieschendorf, und T. Röfer: *Gamecontroller2.* https://github.com/bhuman/GameController, 2012

```
                * Pakete empfangen und in Datenstruktur
                   kopieren
                */
                publishProgress(recv_packet);
                }
        }

        @Override
        protected void onProgressUpdate(DatagramPacket...
           packet) {
                pManager.sendChangedPacket(mGameInfo);
        }
}
```

Die Grundstruktur der Klasse `NetworkReceive.java` und wie sie `AsynchTask` erweitert, sieht man in Listing 13.3. Wenn die Klasse `NetworkReceive` instanziiert wird, dann wird explizit die Methode `execute(Params)` aufgerufen und als Parameter ein leeres String-Objekt übergeben, da keine speziellen Parameter benötigt werden. Execute sorgt dafür, dass die Methode `doInBackground(Params)` ausgeführt wird, die in einer while-true Schleife UDP Pakete verarbeitet. Nachdem ein Paket empfangen und in die passenden Datenstrukturen überführt wurde, wird die Methode `publishProgress(Progress)` aufgerufen. Diese Methode, die von `AsynchTask` bereitgestellt wird, sorgt dafür, dass der Fortschritt, den die `doInBackground` Methode erzielt hat, weiterverarbeitet werden kann. Diese Verarbeitung geschieht in `onProgressUpdate(Progress)`, wo die durch das UDP Paket gefüllte Datenstruktur an eine andere Klasse weitergegeben wird.

PacketManager Observer Model

Die Klasse `PacketManager` erweitert die Klasse `Observable` und setzt damit das Beobachtermodell um. Viele Szenen in der App müssen auf Änderungen im Spielgeschehen reagieren, zum Beispiel werden Roboter durch den Schiedsrichter bestraft oder wenn ein Tor gefallen ist, wird dieses in der App visualisiert. Durch eine zentrale Instanz, den `PacketManager`, ist es möglich, dass sich die einzelnen Szenen beliebig als Beobachter an- und abmelden. So werden die angemeldeten Szenen bei neuen Paketen rechtzeitig informiert und können auf Veränderungen im Spiel schnell reagieren.

Konzept von SmartRef und Verwendung der AndEngine

Die AndEngine benutzt das Konzept der Szenen, wobei man sich eine Szene wie ein Level in einem Spiel vorstellen kann. Eine Szene setzt sich aus verschiedenen Objekten zusammen. Das kleinste gemeinsame Objekt, welches sich alle Objekte teilen, ist das Entity Objekt. Dieses Objekt verwaltet die grundlegensten Daten wie Position, Rotation, Farbe, Eltern-Kind Beziehung zu anderen Objekten und einige mehr. Die Unterklassen einer Entity Klasse sind dann die Objekte, mit denen man eine Szene füllen kann wie zum Beispiel Sprite, Text, Texture und Body Objekte. Eine Szene selbst ist auch ein Entity Objekt und

stellt den grundlegenden Rahmen für alles weitere dar. Sie ist Elternknoten für alle Objekte, die einer Szene hinzugefügt werden. Es entsteht somit ein Szenengraph, wie er auch schon aus Java3D bekannt ist. SmartRef verwendet die Szenen, um die einzelnen Zustände INITIAL, READY, SET, PLAYING und FINISHED des GameControllers und somit auch des Roboters darzustellen. Für jeden Zustand, in dem sich der Roboter befinden kann, gibt es in der SmartRef-App eine Szene, wobei die Zustände READY und SET zusammengefasst wurden. Dies ist geschehen, weil sich bei der Analyse der Zustände herausstellte, dass sich die Beiden in Bezug auf den Informationsgehalt, nicht unterscheiden. In den Zuständen READY und SET sind nur die Positionen der Roboter auf dem Feld entscheidend, falls diese manuell positioniert werden müssen. Die Information, dass die Roboter in Set stehen bleiben müssen, ist nur für die Roboter, aber nicht für die Schiedsrichter von Belang.

Verwaltung und Aufbau der Szenen

Um die Verwaltung der Szenen übersichtlicher zu gestalten, gibt es in der SmartRef-App eine Klasse, die sich um den Wechsel der Szenen untereinander kümmert. Durch den zentralen Punkt der Szenenverwaltung ist es möglich, gezielt zwischen einzelnen Szenen zu wechseln. Zusätzlich zu den einzelnen Szenen verwaltet der `SceneManager` einige szenenübergreifende Variablen. Der Aufbau einer Szene an sich erfolgt nach dem folgenden Schema. Im Konstruktor der Szene werden globale AndEngine Objekte übergeben, damit jede Szene auf dieselbe Kamera, Engine oder Physik zugreifen kann. Das Laden der Ressourcen in den Speicher des Smartphones geschieht getrennt von dem Platzieren der Objekte in der Szene. Das Platzieren von Objekten geschieht in verschiedenen Layern. Jeder Layer ist ein Entity-Objekt, welches, wie schon im vorherigen Abschnitt genannt, nur die grundlegensten Informationen enthält, die die AndEngine benötigt. Da sich zwei grafische Objekte untereinander verdecken können, es also passieren kann, dass Objekt A über Objekt B liegt oder Objekt B über Objekt A, wurden die Layer eingeführt. In der AndEngine wird Objekt B von Objekt A verdeckt wenn Objekt A nach Objekt B der Szene hinzugefügt wurde. Es ist also wichtig die Reihenfolge zu beachten in der die Objekte der Szene hinzugefügt werden. Mit Layern ist es möglich die Überdeckung der Objekte gezielt zu steuern. In der SmartRef App werden drei Entity Objekte (`layerBottom, layerMiddle, layerTop`) erstellt und sie werden in der genannten Reihenfolge dem Scene Objekt als Kinder hinzugefügt. Diesen drei Layern kann man wiederum neue Kinder hinzufügen, die dann die Eigenschaft haben, dass Objekte im mittleren Layer immer vor Objekten im unteren Layer liegen und diese Objekte somit verdecken werden.

Die Trennung von Laden und Platzieren hat den Vorteil, dass man die Ressourcen schon vor dem Anzeigen der eigentlichen Szene laden kann. Das kann unter anderem in einem extra Ladebildschirm geschehen, falls die Anzahl der zu ladenden Ressourcen groß ist. So wird dem Benutzer nur die fertige Szene präsentiert und es wird verhindert, dass der Benutzer glaubt die App würde nicht mehr reagieren.

Ressourcen in der AndEngine

Ein paar der für SmartRef wichtigsten Ressourcen der AndEngine werden im Folgenden beschrieben.

FONT

Ein Font Objekt in der AndEngine ist die Grundlage für das Textobjekt. Es definiert Aussehen, Farbe und Größe der Schrift.

TEXT

In einem Text Objekt ist es möglich, die angelegte Schriftart zu verwenden, um Text auf dem Bildschirm auszugeben. Text Objekte benötigen unter anderem eine initiale Position und Text, welche aber zur Laufzeit verändert werden können.

TEXTURE

Eine Textur ist ein Bild, welches man in der App benutzen möchte. In der AndEngine gibt es zwei wichtige Klassen, um die Bilder, die in einem Ordner sind, in den Speicher des Smartphones zu bekommen. Zunächst gibt es die `BitmapTextureAtlas` Klasse. Ein Textur Atlas ist ähnlich einem leeren Blatt Papier, welches durch Parameter in seiner Größe begrenzt ist. Ist die Größe erst einmal festgelegt, kann man innerhalb dieser Fläche seine Bilder positionieren. Es ist also wie ein Atlas mit den Bildern als Ländern und den Koordinaten der Bilder als Positionen auf der Fläche. Die Positionierung erfolgt über `ITextureRegion` Objekte. Diese Objekte legen die Position der einzelnen Bilder in dem `BitmapTextureAtlas` fest. Dabei ist es auch möglich, dass die Textur Atlas Größe mit der des `ITextureRegion` Objekts überein stimmt.

SPRITE

Ein Sprite ist im klassischen Sinn eine 2D Grafik oder Animation, die sich vor einem Hintergrund bewegt. In modernen Spielen werden Sprites auch in 3D Umgebungen verwendet, um Rauch, Feuer, Gras oder ähnliches, wie in Abbildung 13.1, darzustellen. So wird ein Sprite in vielen Situationen verwendet, wie auch in der SmartRef App. Dort werden Sprites nicht nur für bewegte Objekte benutzt, sondern auch für Hintergründe oder Buttons. Da `ITextureRegion` Objekte in der AndEngine die Bildinformationen beinhalten, können `ITextureRegion` Objekte verwendet werden, um Sprites zu erzeugen.

Listing 13.4: Sprite Objekt

```
Sprite demoSprite = new Sprite(pX, pY, pITextureRegion,
        pEngine.getVertexBufferObjectManager());
```

Das `ITextureRegion` Objekt beinhaltet, wie schon oben genannt, die Position der Grafik im Textur Atlas. Zusätzlich gibt man noch die Position des Sprites auf dem Bildschirm und den `VertexBufferObjectManager` der AndEngine als

Parameter an. Der `VertexBufferObjectManager` speichert Koordinaten der Sprites in einem Array und hält das Array im Grafikspeicher, um die Performance, im Gegensatz zum Hauptspeicher, zu verbessern.

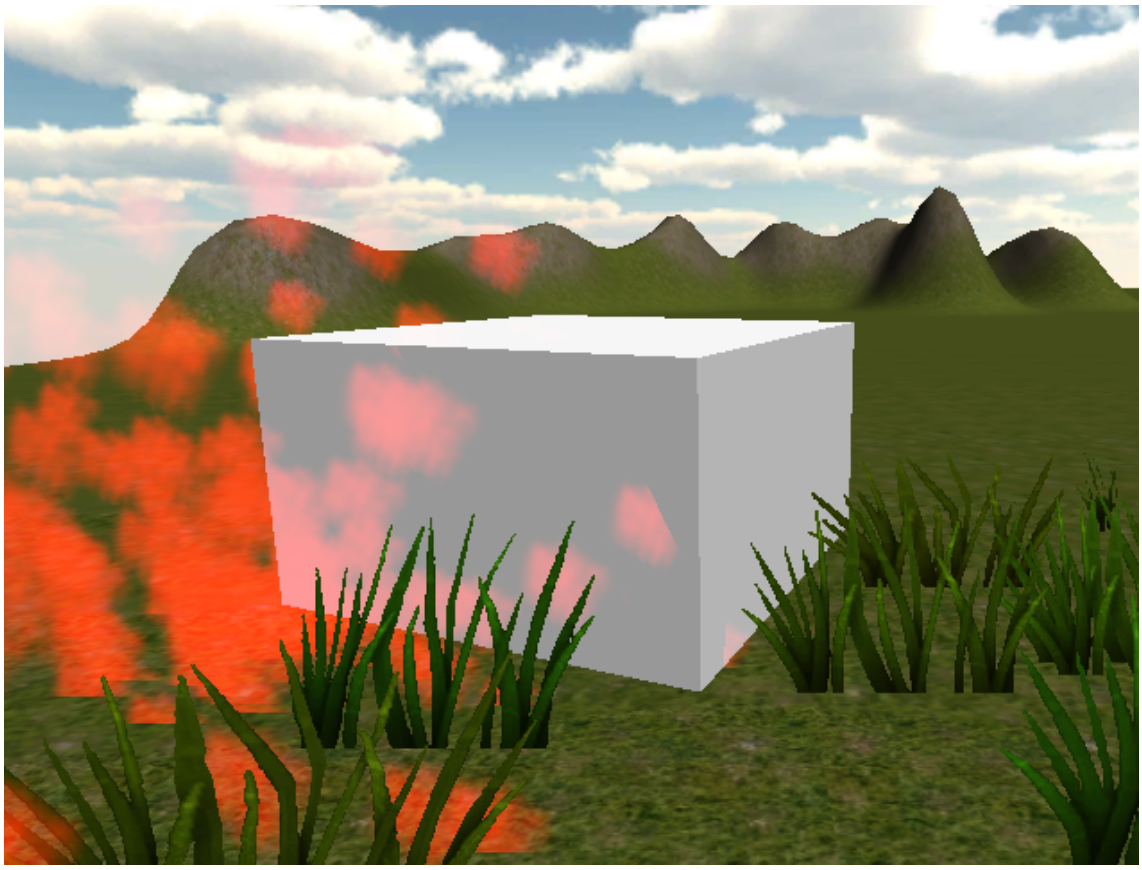


Abbildung 13.3: Gras- und Partikelsprites in einer 3D Umgebung

Box2D

Box2D[5] ist eine Open Source Physik Engine, die sich auf physikalische Wechselwirkung von starren Körpern im zweidimensionalen Raum spezialisiert hat. Die Engine wird seit 2007 in C++ entwickelt und kam unter anderem in Angry Birds zum Einsatz. Mittlerweile ist Box2D neben Java in viele andere Sprachen portiert worden. Box2D wird als Erweiterung[6] für die AndEngine in SmartRef benutzt. Um Box2D in der AndEngine verwenden zu können, muss zunächst ein `PhysicsWorld` Object angelegt werden.

Listing 13.5: FixedStepPhysicsWorld Objekt mit Parametern

```
FixedStepPhysicsWorld physicsWorld =
    new FixedStepPhysicsWorld(60, new
        Vector2(-SensorManager.GRAVITY_EARTH * 2, 0f),
        false, 8, 3);
```

[5] [24] Catto, Erin: *Box2d.* http://box2d.org/, 2013

[6] [25] Gramlich, Nicolas: *Andengine physics box2d extension.* https://github.com/nicolasgramlich/AndEnginePhysicsBox2DExtension, 2013

Dieses Objekt legt den physikalischen Rahmen fest, in dem sich alle Objekte der Szenen bewegen werden. Der erste Parameter legt die Anzahl der Berechnungschritte pro Sekunde fest, damit die Physikberechnung nicht abhängig von der Geschwindigkeit des Prozessors im Smartphone ist. Zu den wichtigsten Parametern gehört die Angabe über die simulierte Gravitation, welche in der SmartRef App die doppelte Anziehungskraft der Erde beträgt. Wirken soll die Gravitation nur in x Richtung, deshalb ist der y-Vector2 Parameter null. Die Gravitation wird in der App die Geschwindigkeit bestimmen, mit der die Penalty Sprites (siehe 13.6b) sich über den Bildschirm bewegen. Der dritte Parameter bewirkt, dass Objekte nicht eigenständig ihr physikalisches Verhalten deaktivieren können, wenn sie zur Ruhe gekommen sind. Zuletzt kommen noch zwei Parameter, welche die Genauigkeit der Simulation festlegen. Wie schon genannt, wird die Physik Engine bei den Sprites angewendet, die die Strafen der einzelnen Roboter visualisieren. Damit Objekte, hier Sprites, der Gravitation ausgesetzt sind oder mit anderen Objekten in physikalischer Wechselwirkung stehen können, benötigen diese Körper.

Körper sind in der AndEngine `Body` Objekte von denen es verschiedene Arten gibt. Zunächst sind dort dynamische Körper, die auf externe Kräfte und andere Körper reagieren. Dazu gibt es statische Körper, welche sich nicht bewegen und zuletzt kinematische Körper, die sich bewegen können, aber nicht durch Kräfte oder andere Körper beeinflusst werden.

Ein wichtiger Parameter bei der Erstellung von `Body` Objekten ist das `FixtureDef` Objekt. `FixtureDef` legt die physikalischen Beschaffenheiten eines Körpers fest.

Listing 13.6: Fixture Objekt in der AndEngine

```
FixtureDef Wall = PhysicsFactory.createFixtureDef(pDensity,
   pElasticity, pFriction);
```

Der Parameter `pDensity` beschreibt das Gewicht des Körpers, `pElasticity` wie stark der Körper nachgeben soll und `pFriction` ist die Reibung. Jedem `Body` wird beim Erzeugen eine Beschaffenheit mitgegeben, damit die Physik Engine das gewünschte Verhalten simulieren kann.

Listing 13.7: Body Objekt in der AndEngine

```
Body penaltyBody =
   PhysicsFactory.createBoxBody(pPhysicsWorld, pAreaShape,
   pBodyType, pFixtureDef);
```

In der SmartRef App werden dynamische Körper für die Penalty Sprites und statische Körper für eine unsichtbare Wand benutzt, damit die Penalty Sprites nicht aus dem sichtbaren Bereich „herausfallen “. Wie in Listing 13.7 zu sehen ist, kann man ein Body Object mit vier Parametern erstellen. PhysicsWorld, BodyType und FixtureDef wurden schon im vorherigen Text erläutert. Neu ist der Parameter AreaShape, der die Grenzen des Körpers festlegt. Man sieht auch, dass die Methode `createBoxBody` verwendet wird, was mit der Form der Sprites zusammenhängt. Es ist ebenso möglich `createCircleBody`, `createLineBody`, `createPolygonBody` oder `createTriangulatedBody` zu verwenden, je nachdem welche Form der Körper haben muss. Erstellte `Body` Objekte werden den zugehörigen Sprites mit der Methode `connectBodyandSprite(Body)` zugeordnet.

Die Verwendung von Box2D in der AndEngine hat geholfen, in der kurzen Entwicklungszeit ein visuell ansprechendes Ergebnis zu erzielen. Ohne mathematische Berechnungen oder manuelle Positionsbestimmungen verhalten sich die Penalty-Sprites wie gewünscht. Sprites werden nicht nur nach dem FIFO-Prinzip hinzugefügt und entfernt, sondern auch, an einer beliebigen Position stehend, entfernt. Hinzu kommt, dass die entstandene Lücke mit schon vorhandenen Sprites gefüllt werden soll. Mit der Box2D Erweiterung wurde all dies erzielt, ohne eventuell fehlerträchtige Berechnungen anzustellen. Zudem erreicht man durch die Simulation der Physik einen natürlicheren Effekt, als wenn man einfach Grafiken an Koordinaten verschoben hätte.

Einschränkung durch die Verwendung von AndEngine

Durch die Entscheidung für fertige Komponenten wurden Einschränkungen bei den lauffähigen Geräten gemacht. Die AndEngine setzt mindestens Android 2.2 voraus[7]. Die verwendete Hardware-Beschleunigung für die Grafik und Physik-Berechnung setzt eine GPU mit OpenGL ES2.0 voraus, dieses ist bei den meisten SoCs (System on a Chip) der Fall[8] und 99,8 % der Android-Geräte[9]. Mit diesen Einschränkungen kann die App mit Stand vom 8. Juli 2013 auf fast 98,7 % der Android-Geräte[10] verwendet werden . Die Unterstützung für das Senden und Empfangen von UDP-Paketen wird von manchen Herstellern leider im Kernel deaktiviert und müsste dann nachträglich aktiviert werden.

Beschreibung der einzelnen Szenen

Im Folgenden werden die einzelnen Szenen der Anwendung beschrieben. Der Ablauf der Beschreibung entspricht dem Ablauf eines normalen Spielverlaufs.

Nach dem Start der App gelangt man in die „Menü Szene“ (siehe Abb. 13.4a). Das Menü bietet die Auswahl für die beiden Benutzermodi der App. Der eine Modus ist für den Hauptschiedsrichter beziehungsweise head-referee der andere für die Assistenten beziehungsweise assistant-referees. Die App verhält sich für die Rollen unterschiedlich, das genaue Verhalten wird bei den entsprechenden Szenen vorgestellt. Wird in der App der „Zurück“-Knopf des Geräts betätigt, gelangt man wieder ins Menü. Beim Übergang zwischen den einzelnen Spielsituationen vibriert das Gerät kurz.

Nachdem man im Menü eine Schiedsrichter-Rolle ausgewählt hat, erscheint die „Initial Szene“. In dieser Szene wird gewartet, bis der GameController in die Vorbereitungsszenen für das Spiel übergeht. Wenn eine Verbindung zum GameController besteht, dann wird die aktuelle Halbzeit angezeigt.

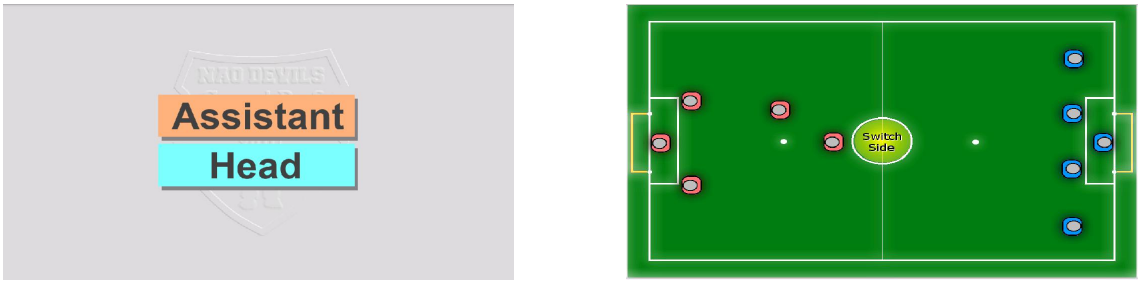
---

[7] [22] Gramlich, Nicolas: *Andengine*. https://github.com/nicolasgramlich/AndEngine/, 2013

[8] [26] Group, Khronos: *Opengl es 2.x - for programmable hardware*. http://www.khronos.org/opengles/2_X/, Mai 2013

[9] [27] Google: *Android dashboards*. http://developer.android.com/about/dashboards/index.html, Juli 2013

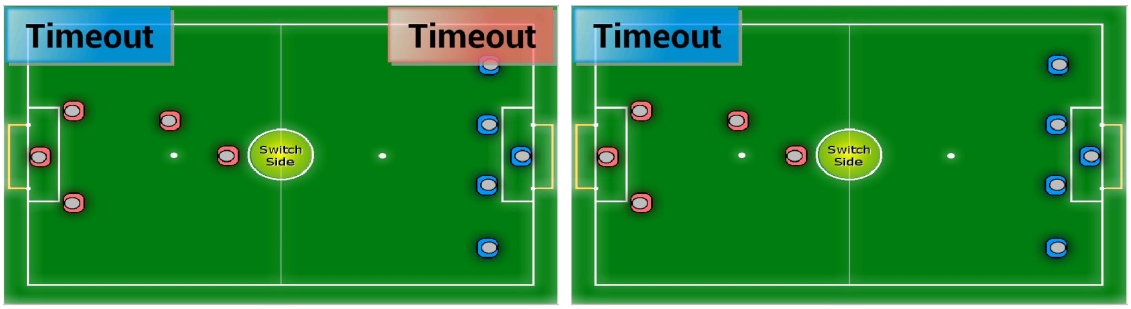
[10] [27] Google: *Android dashboards*. http://developer.android.com/about/dashboards/index.html, Juli 2013



(a) Menü der SmartRef App **(b)** Ansicht für Assistenten: Anstoß für Rot

Abbildung 13.4: Menü und ReadySet-Szene

Sobald der GameController das Einlaufen mit der Zustandsänderung zu „Ready “ startet, gelangt man in die „ReadySet Szene“ (siehe Abb. 13.4b). In diese Szene gelangt man auch nach einem Tor, weil die Roboter zurück auf ihre Startpositionen laufen. Nach dem Einlaufen geht der GameController automatisch in den Zustand „Set“ über. Für die Schiedsrichter gibt es in beiden Spielzuständen keine Unterschiede, weshalb diese zusammengefasst wurden. Im Hintergrund werden die offiziellen Startaufstellungen angezeigt. Auf diese Positionen werden die Roboter von den Schiedsrichtern gestellt, falls sie sich nicht regelkonform positionieren. Anhand der gesendeten Informationen des GameControllers kann das anstoßende Team für die Anzeige bestimmt werden. Bei der Berührung des Bildschirms wird die Aufstellung gespiegelt, damit ein Schiedsrichter, der auf der nicht angezeigten Seite des Felds steht, eine passende Ansicht bekommt. Für den Hauptschiedsrichter werden zusätzlich die möglichen Auszeiten der Mannschaften angezeigt (siehe Abb. 13.5a). Falls ein Team keine Auszeit mehr hat, wird der Knopf ausgeblendet (siehe Abb. 13.5b). Die Knöpfe für die Auszeit werden nur in den Situationen angezeigt, in denen eine Mannschaft diese nach den Regeln auch nehmen darf.



(a) Ansicht für Hauptschiedsrichter: Anstoß für Rot, beide Auszeiten vorhanden **(b)** Ansicht für Hauptschiedsrichter: Anstoß für Rot, Rot hat seine Auszeit bereits genutzt

Abbildung 13.5: Anzeige der ReadySet-Szene mit und ohne Timeout

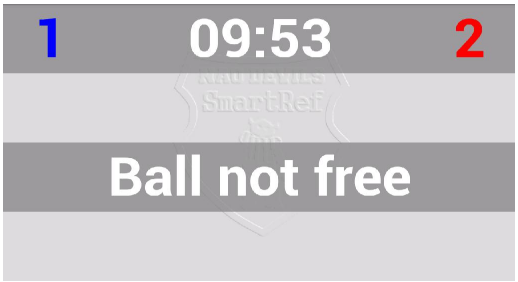
Bei Betätigung des „Timeout“-Knopfs gelangt man in die „TimeOut Szene“ (siehe Abb. 13.6a). Das Signal für eine Auszeit wird vom GameController nicht gesendet, deshalb wird ein manueller Eingriff benötigt. Es wird die verbleibende Zeit der Auszeit angezeigt und das Team welches diese genommen hat. Durch Berühren der „Timeout“-Anzeige kann die Auszeit vorzeitig beendet werden. Sobald von den fünf Minuten Auszeit nur noch 30 Sekunden verbleiben, blinkt die Uhr und das Gerät vibriert zyklisch.

(a) Anzeige einer Auszeit für das rote Team

(b) Normale Spielsituation, Spieler 4 von Rot hat noch 9 Sekunden eine Strafe, Spieler 3 von Blau hat noch 13 Sekunden Strafe, für den roten Spieler 5 wurde ein Pickup gerufen.

Abbildung 13.6: TimeOut-Szene und eine Ansicht der Playing-Szene

Wenn das Spiel vom GameController gestartet wurde, geht die Anwendung in die Hauptanzeige, die „Playing Szene“ (siehe Abb. 13.6b), über. Die „Playing Szene“ besteht aus drei wichtigen Elementen, der „Gameinfobar“, den „Penalty-Tils“ und der „Eventinfobar “. Die „Gameinfobar“ zeigt die aktuelle Zeit und den Spielstand an. 30 Sekunden vor dem Halbzeitende blinkt die Zeit und für den Hauptschiedsrichter wird eine haptische Rückgabe über eine zyklische Vibration gegeben. Die „Penalty-Tils“ zeigen die aktuellen Strafen für die Roboter an. Die Farbe der Kachel gibt das Team an, dann beinhaltet sie noch die Spielernummer und die Strafzeit. Beim Erzeugen einer Kachel gibt es eine kurze Vibration für den Assistenzschiedsrichter, damit im signalisiert wird ob ein Spieler vom Platz genommen werden muss. Für den Assistenzschiedsrichter gibt es sieben Sekunden vor dem Ablaufen einer Strafe eine Rückmeldung über Vibration, damit der Roboter zur Rückkehr positioniert werden kann. Soll ein Roboter wegen eines Defekts aufgenommen werden, erscheint eine „Pickup“-Kachel. Die „Pickup“-Kachel verschwindet automatisch nach zehn Sekunden, alle anderen erst wenn der Spielleiter auch die Roboter wieder freigegeben hat. Beim Verschwinden einer Kachel rutschen die verbleibenden Kacheln automatisch nach.



(a) Ansicht für Hauptschiedsrichter: Die Eventinfobar zeigt an, ob der Ball vom nicht anstoßenden Team gespielt werden darf.

(b) Ansicht für Assistenten: Die Eventinfobar zeigt an, dass der Ball vom blauen Team ins Aus gespielt wurde.

Abbildung 13.7: Unterschiedliche Anzeigen der Eventinfobar

Die „Eventinfobar“ zeigt wichtige Ereignisse für die Schiedsrichter an. Nach dem Anpfiff darf für zehn Sekunden das verteidigende Team den Mittelkreis nicht betreten. Diese Zeitspanne wird dem Hauptschiedsrichter über dauerhafte Vibration und der Anzeige im Bildschirm signalisiert (siehe Abb. 13.7a). Der Assistentsschiedsrichter erhält die Information

von welchem Team ein Ball ins Aus geschossen wurde (siehe Abb. 13.7b). Bei Berührung des Informationsbalken bekommt der Assistenzschiedsrichter einen Regelhinweis für die Einwurfpositionen gezeigt (siehe Abb. 13.8). Für den Einwurf gibt es bestimmte Regeln. Der Roboter, der den Ball ins Aus geschossen hat, darf hieraus keinen Vorteil ziehen. Entweder muss der Ball einen Meter hinter dem Spieler, der ihn zuletzt berührte, oder einen Meter hinter der Aus-Position platziert werden,in Abhängigkeit davon, welche Position weiter vom Roboter entfernt ist. In der Grafik ist für jeden Roboter ein passend farbener Ball für alle Möglichkeiten eingezeichnet.

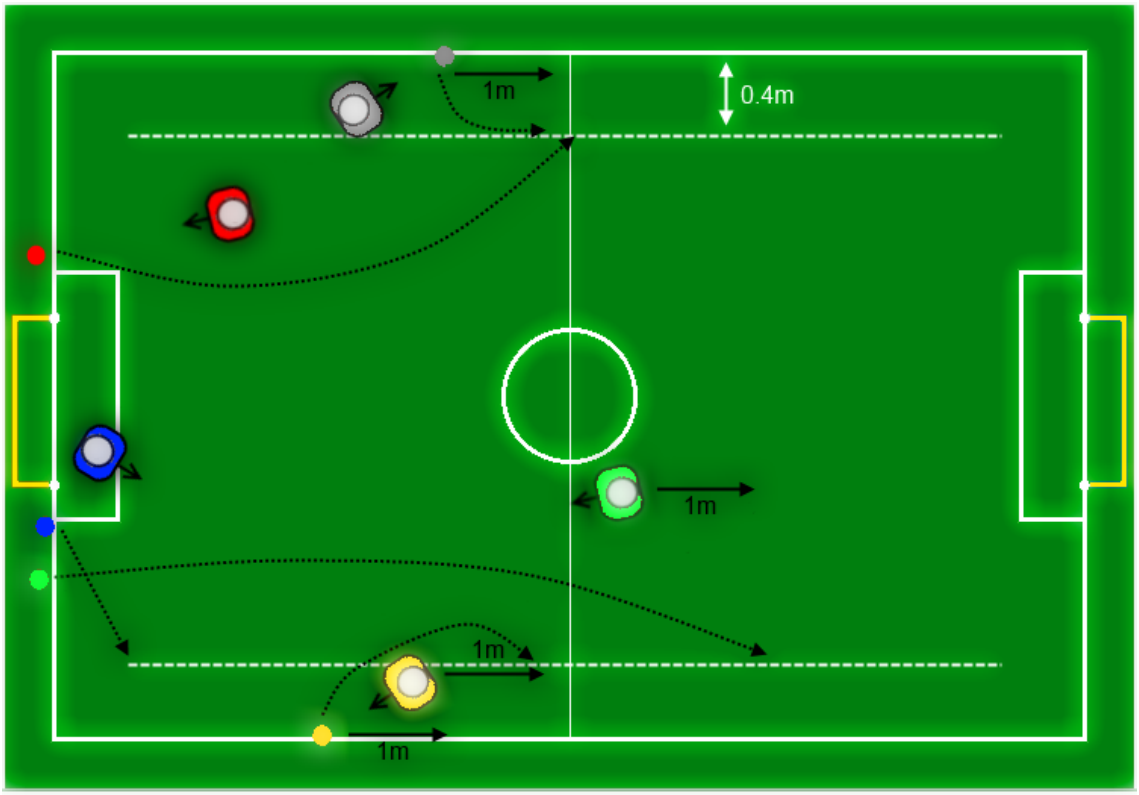


Abbildung 13.8: Ansicht für Assistenten: Spickzettel für die Platzierung des Balles. Jeder Spieler gehört zu dem Ball seiner Farbe. Die schwarzen gepunkteten Linien zeigen die Position an, wohin der Ball zurück gelegt werden muss. Der schwarze Pfeil zeigt den Stand zum Spieler bzw. der Ausposition. Die weiße gestichelte Linie zeigt die nach den Regeln vorgeschriebene Linie für das Zurücklegen.

Wenn die Zeit einer Halbzeit abgelaufen ist, wird die „Finished Szene“ angezeigt. In der Halbzeit wird die verbleibende Zeit der Pause angezeigt, vor Ablauf der Zeit werden die Schiedsrichter durch Vibration und blinken informiert.

SmartRef beim Robocup 2013

Noch vor den ersten offiziellen Spielen wurde der Wettbewerb der OpenChallenge abgehalten. In der Halle gab es erhebliche Probleme mit dem WLAN, die Pakete des GameControllers sind mit starker Verzögerung angekommen. Durch die Anwendung konnte diese Verzögerung gut visualisiert werden, teilweise sind die Informationen erst nach einer Minute Verzögerung angekommen. Um die Verzögerung über die offiziellen Access Point zu

umgehen, musste eine direkte WLAN-Verbindung zwischen einem Laptop und dem Präsentationsgerät aufgebaut werden. Im Vergleich zu den Tests in Dortmund gab es noch immer eine Verzögerung von ca. zehn Sekunden. Trotz der Verzögerung konnte die SmartRef-Anwendung bei der Präsentation vollständig vorgestellt werden und hat positive Resonanz hervorgerufen. Bei der offiziellen Bewertung[11] wurde der vierte Platz erreicht. Mehrere Teams haben die Idee gewürdigt und einen Bedarf für die Unterstützung der Schiedsrichter gesehen. Das Team RoboCanes von der University of Miami hat um ein Testexemplar gebeten. Eine offizielle Ausrüstung der Schiedsrichter bei den Turnieren ist durch die weiterhin zu erwartenden WLAN-Probleme unwahrscheinlich.

[11] [28] RoboCup SPL Technical Committee: *Robocup 2013 - spl - results*. http://www.tzi.de/spl/bin/view/Website/Results2013, 2013

Kapitel 14

Kalibrierung

Die andauernde Belastung der Naos durch Tests oder Spiele wirkt sich häufig extrem auf den Zustand der Roboter aus. Die Gelenkwinkel verschleißen und nehmen damit direkten Einfluss auf den Lauf der Naos. Um einen guten Lauf zu garantieren, müssen die Gelenkwinkel und der Lauf immer wieder aufeinander abgestimmt werden. Diesen Vorgang nennt man *JointCalibration*. Wird die Grundstellung der Gelenke verändert, hat dies auch Einfluss auf die Kamera-Kalibrierung. Nach jeder *JointCalibration* sollte daher auch die Kalibrierung der Kamera erneut durchgeführt werden. Zunächst wird daher die Gelenkwinkel-Kalibrierung beschrieben und anschließend wird auf die Kamera-Kalibrierung eingegangen.

14.1 Joint-Kalibrierung

Als limitierender Faktor für den Erfolg in Test- als auch Ligaspielen, hat sich eine falsch konfigurierte Gelenkwinkelkalibrierung der Roboter erwiesen. De facto kann kein Spiel gewonnen werden, wenn die Naos regelmäßig umfallen oder schlichtweg zu langsam laufen. Jeder Nao besitzt pro Fuß allein fünf Motoren und Gelenke die auf Grund unterschiedlicher Belastungen während der Spiele, einen undefinierbaren Verschleiß aufweisen. Dieser Verschleiß kann verschiedenste Ursachen, wie beispielsweise mechanische Spielräume, die Abnutzung der eingebauten Motoren oder der Getriebe haben. Der Grad und die genaue Ursache kann daher nicht immer exakt bestimmt werden.

An aktuelle Ansätzen, um aus den niedrigsten Strömen in den Gelenken eine ideale Position des Naos zu errechnen, wird zur Zeit intensiv geforscht. Eine Diplomarbeit in diesem Bereich kann in Zukunft möglicherweise einige Arbeit vor den Spielen ersparen. Bis dahin müssen die Gelenkwinkel vor jedem Spiel so kalibriert also angepasst werden, dass für jeden einzelnen Roboter eine ideale Ausgangslage geschaffen wird. Der individuelle Verschleiß verhindert, alle Roboter mit der gleichen Kalibrierung auszustatten. So muss jeder Nao für einen optimalen Lauf, individuell betrachtet und entsprechend kalibriert werden.

14.1.1 Parameter und Umgebung

Die Kalibrierung beginnt direkt auf dem Spielfeld, auf dem der Nao später spielen wird. Jeder Untergrund weißt eine andere Beschaffenheit auf, welche sich entsprechend auf den Lauf auswirkt. Bei Turnieren können die Naos daher auch nicht vorher auf dem heimischen Feld kalibriert werden, besser sollte dies vor Ort und wenn möglich, vor jedem Spiel geschehen.

Die Kalibrierung selbst, findet mit Hilfe des Simulators statt. Wozu man die *RemoteRobot* Szene lädt und sich beispielsweise über LAN mit dem Nao verbindet.

Zwei Befehle sollten geladen werden:

- `get representation:MotionRequest`
- `get representation:JointCalibration`

Als Antwort erhält man jeweils den *set* Befehl inklusive der aktuellen Konfiguration.

14.1.2 Vorkonfiguration

Das Vorgehen kann als empirisch betrachtet werden. In einem ersten Schritt wird der Nao auf Augenhöhe angehoben, sodass die Stellung der Fußwinkel betrachtet werden kann. Wie Abbildung 14.1 verdeutlicht, sollten die die Füße in einem 90° zum Bein stehen, also völlig senkrecht. Außerdem sollten beide Füße völlig glatt auf einer Fläche stehen.

Fußwinkel Pitch Wird der Nao seitlich gehalten, kann schnell erkannt werden ob einer der Füße in einem anderen Winkel zum Untergrund steht. Dieser Schritt scheint sehr trivial, ist aber extrem ausschlaggebend für einen stabilen Lauf.

Wird eine Fehlstellung festgestellt muss diese korrigiert werden. Dazu einfach den `set representation:JointCalibration` Befehl entsprechend anpassen und ausführen. Der Nao korrigiert seine Gelenke entsprechend. Geändert wird immer der Offset des entsprechenden Gelenkes. Die Gelenk-Winkel zählen von unten nach oben und können ebenfalls leicht gefunden werden, indem einfach testweise ein extremer Winkel angegeben wird. So kann sehr schnell das entsprechende Gelenk lokalisiert werden.

Fußwinkel Yaw Die gleiche Anpassung wird für den seitlichen Winkel des Knöchels (Yaw) ausgeführt. Abbildung 14.2 zeigt, wie die Gelenke stehen sollen.

Beinlänge Nach der Anpassung der Fußwinkel fällt manchmal auf, dass ein Bein höher oder Tiefer als das andere steht. In diesem Fall muss nach eigenem Ermessen entweder das eine Bein angehoben oder entsprechend das Andere abgesenkt werden. Haben die Beine unterschiedliche Längen, steht der Nao schief und läuft entsprechend instabil, da er mit

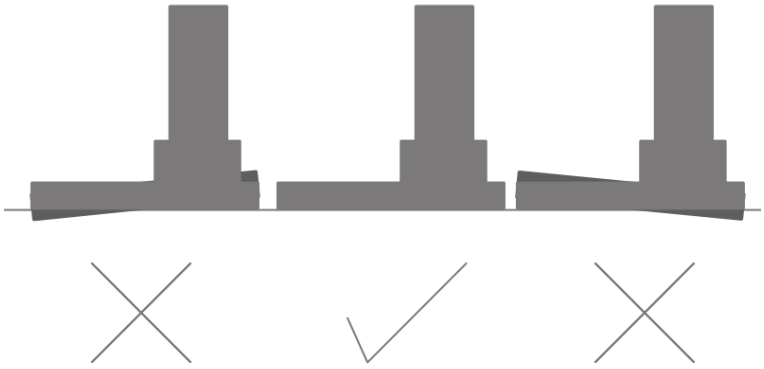


Abbildung 14.1: Seitenansicht

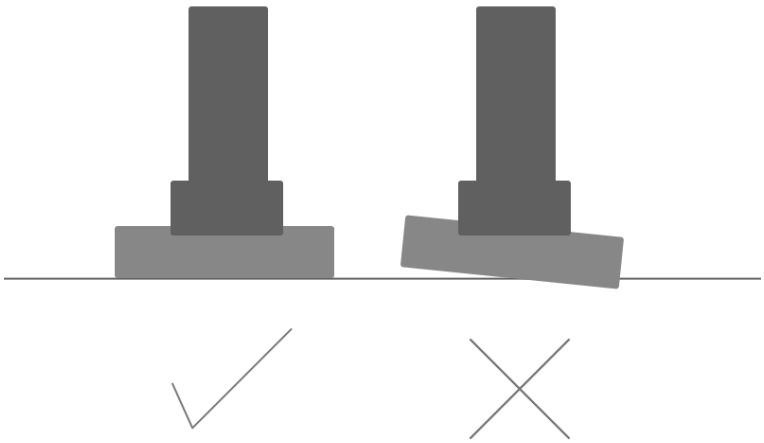


Abbildung 14.2: Frontalansicht

einem Bein eher in den Boden treten wird. Dies führt dazu, dass sich der Nao aufschaukelt und sehr wahrscheinlich bei höheren Geschwindigkeiten umfällt.

Oberkörper Je nach Lauf muss gegebenenfalls der Oberkörper angepasst werden. Dazu können die Hüftgelenke entsprechend korrigiert werden. Auf den letzten Turnieren hat sich eine leichte Neigung des Oberkörpers nach Vorne als förderlich für einen stabilen Lauf erwiesen. Liegt der Schwerpunkt des Naos allerdings zu weit vorne, tritt er in den Boden und wird instabil. Dieses Fehlverhalten ist offensichtlich und leicht zu erkennen. Um dieses Problem zu vermeiden, kann mit verschiedenen Kombinationen aus Hüft- und

Fuß-Pitch-Gelenkstellungen experimentiert werden. Eine minimale Verlagerung des Oberkörpers wirkt sich häufig extrem auf den Lauf aus.

14.1.3 Lauftests

Letztendlich zählt nicht wie die Gelenke des Naos rein optisch wirken, sondern ob die gewählte Stellung einen guten Lauf ermöglicht. Abbildung 14.3 zeigt, welche Tests sich im Eifer der Turniervorbereitungen als nützlich erwiesen haben.

Tippeln und geradeaus Generell sollte der Nao erst mal ohne umzukippen oder stark zu schwanken auf der Stelle tippeln können. Dazu muss der MotionRequest auf `motion = walk` gestellt werden und in X Richtung ein kleiner Wert z.B. 1 angegeben werden. Tippelt der Nao stabil auf einer Stelle sollte der X Wert erhöht werden. Im Idealfall läuft der Nao auf 200 noch stabil und geradeaus. Minimale Abweichungen nach Links oder Rechts sind tolerierbar. Läuft der Nao erkennbar eine Kurve, stehen die Füße häufig falsch und treten unterschiedlich in den Boden, was zu einer Drehung führt. An dieser Stelle muss nachkalibriert werden, wobei keine allgemeingültige Lösung vorliegt. Zwangsläufig müssen verschiedene minimale Änderungen immer wieder getestet werden, bis der Lauf stabil wirkt. Wird mit `get representation:MotionRequest` zusätzlich ein weiterer `set-Befehl` geladen, wobei dieser mit `motion = specialAction` und der SpecialAction `StandStraigt` ausgeführt wird, kann schnell zwischen laufen und stehen gewechselt werden.

Seitlich Der Nao sollte außerdem seitlich laufen können, ohne dabei große Abweichungen nach vorne oder hinten zu haben. Dazu sollte im MotionRequest ein kleines Y gewählt werden und keine Rotation eingetragen sein.

Im Kreis Weiterhin sollte der Nao nahezu einen perfekten Kreis laufen können. Wie man Abbildung 14.3 entnehmen kann, sollte der Laufweg schon im Ansatz einem Kreis entsprechen. Bricht der Nao zu sehr aus, sollten verschiedene Parameter verändert werden, bis die gelaufene Kurve ansatzweise passt. Diese Konfiguration ist wichtig, da sich der Nao teilweise wie getestet dem Ball annähert. Die Blickrichtung sollte dabei variiert werden, also einmal Blick in den Kreis, einmal Blick nach Außen. Dazu sollte ein kleines Y gewählt werden, sowie eine leichte Rotation. Die Rotation sollte je nach gewünschter Blickrichtung angepasst werden.

Es kommt immer wieder vor, dass ein Nao einfach nicht mehr gut kalibriert werden kann, da die Gelenke einfach schon zu verschlissen sind. Bei den letzten Turnieren kam es mehrfach vor, dass die Kalibrierung einzelner Roboter zu keinem guten Ergebnis führte. Letztendlich stellte sich heraus, dass einige Gelenke einfach schon defekt waren und dies nicht vor den Lauftests geprüft wurde. Es macht daher Sinn, vor der eigentlichen Kalibrierung, die Gelenke einmal generell auf etwaige Beschädigungen zu prüfen. Bewegt man die Gelenke von Hand und spürt abrupte Bewegungen im Gelenk oder fehlt einfach der Widerstand den das Getriebe sonst aufbaut, ist ein Gelenk häufig defekt. Diese Tests kann man gut parallel mit einem weiteren Nao durchführen. So bekommt der Tester schnell ein Gefühl für den aktuellen Zustand des Raboters.

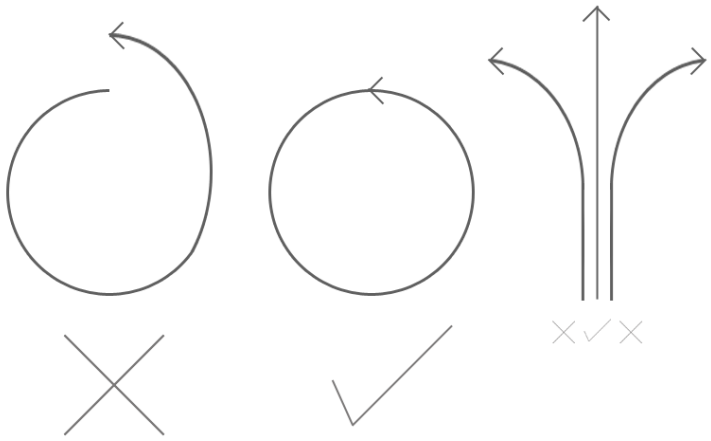


Abbildung 14.3: Laufwege

14.1.4 Speichern

In */Config/Robots* gibt es für jeden Roboter eine *jointCalibration.cfg* in welche die entsprechenden Werte eingetragen werden müssen. Liegt die Konfigurationsdatei nicht vor, werden allgemeine Werte aus dem *Default* Ordner verwendet. Die Änderungen müssen natürlich in das Repository eingetragen werden.

14.2 Kamera-Kalibrierung

bearbeitet von: Bianca Patro, Heinricht Pettenpohl

Die Kamera-Kalibrierung ist wichtig, damit die Roboter die Linien und den Ball zuverlässig erkennen können und die erkannten Linien auch mit den berechneten Positionen übereinstimmen. Der Nao verfügt über eine obere und eine untere Kamera (siehe Abbildung 14.4) Die Kameras sind an einer bestimmten Stelle im Kopf angebracht. Die genauen Positionen können sich aber durch Stürze oder einfach nur Aufstehen nach dem Start immer wieder minimal verändern. Bei der Kamera-Kalibrierung werden die Werte der Kamera-Matrizen so gesetzt, dass die Positionen wieder richtig übereinstimmen und der Roboter die Linien auch dort erkennt, wo sie wirklich sind.

Im nächsten Abschnitt erfolgt zuerst eine kurze Beschreibung der Parameter, die man bei der Kalibrierung einstellen kann. Danach folgt ein Abschnitt, der den Ablauf der Kalibrierung beschreibt. Der dritte Abschnitt zeigt dann an Beispielen, wie sich die Veränderungen der Parameter auswirken und wie ein gutes Resultat der Kamera-Kalibrierung aussieht. Zuletzt wird noch der auf dem RoboCup entwickelte Calibration-Helper kurz vorgestellt.

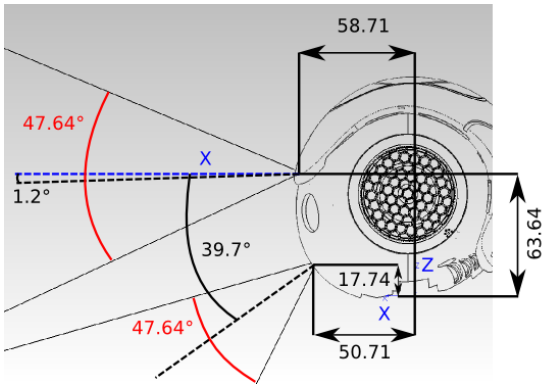


Abbildung 14.4: Skizze der Kamerapositionen im Kopf des Naos [1]

14.2.1 Parameter

Die Parameter für die Kamera-Matrizen bestehen grundlegend aus den drei Parametern Pan, Tilt und Roll. Also das Schwenken nach links und rechts, die Neigung nach oben und unten, sowie das Rollen nach links und rechts. Diese drei Parameter gibt es jeweils für die untere und die obere Kamera sowie für den ganzen Körper, welcher Einfluss auf beide Kameras hat. Dementsprechend besteht die `representation:CameraCalibration` aus den folgenden Parametern:

```
cameraLowerTiltCorrection = 0.00;
cameraLowerRollCorrection = 0.00;
cameraLowerPanCorrection = 0.00;
cameraUpperTiltCorrection = 0.00;
cameraUpperRollCorrection = 0.00;
cameraUpperPanCorrection = 0.00;
bodyTiltCorrection = 0.00;
bodyRollCorrection = 0.00;
```

Der Parameter `bodyPanCorrection` ist nicht erforderlich, da dies eine einfache Rotation des Roboters ist. Er wird über die `representation:RobotPose` entsprechend angepasst.

```
rotation = 1.602;
translation = {x = 0; y = -3000;};
```

Der Parameter `translation` gibt an, wo der Roboter sich auf dem Spielfeld befindet. `X` ist dabei die Abweichung von der Mittellinie und `Y` ist die Abweichung von der Seitenlinie. Dabei ist (0, -3000) genau die Kreuzung der beiden Linien. Die Rotation von 1.602 Radian entspricht einer Drehung des Roboters zum Mittelkreis.

Abbildung 14.5: Der Roboter muss mittig auf dem Kreuzungspunkt stehen.

14.2.2 Ablauf der Kalibrierung

Der Roboter wird auf die Kreuzung zwischen Seitenlinie und Mittellinie mit Blickrichtung zum Mittelkreis gestellt(siehe Abbildung 14.5). Dabei ist es wichtig, dass der Roboter genau mittig auf den Kreuzungspunkt steht und zum Mittelkreis gedreht ist.

Mit der Simulatorszene RemoteRobot wird eine Verbindung zu dem Roboter hergestellt. Dort wird dann der Befehl `call CameraCalibration` ausgeführt. Die Ausgabe in der Simulatorkonsole sieht dann wie folgt aus.

Listing 14.1: Simulatorausgabe nach Ausführung des Befehls `call CameraCalibration`

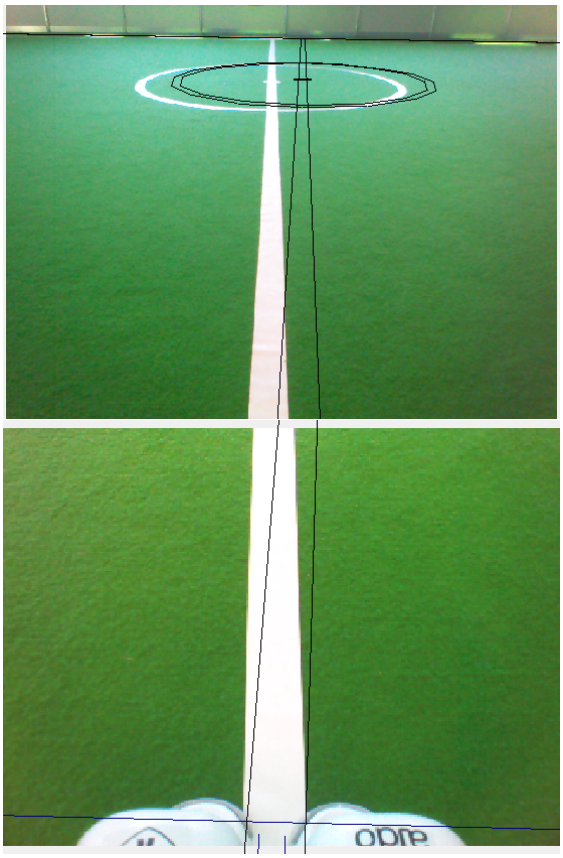
```
1 call CameraCalibration
2 set representation:MotionRequest motion = specialAction;
    standType = doubleSupport; specialActionRequest =
    {specialAction = stand; mirror = false;}; walkRequest =
    { request = { rotation = 0; translation = {x = 0; y =
    0;}; }; requestType = speed; }; kickRequest = {
    kickTarget = {x = 0; y = 0;}; }; kickDirection = 0;
    kickTime = 0;
3 set representation:RobotPose rotation = 1.602; translation
    = {x = 0; y = -3000;}; validity = 0.2;
4 set representation:HeadControlRequest controlType =
    direct; pan = 0; tilt = -0.3;
5 set representation:HeadControlRequest controlType =
    direct; pan = 1; tilt = -0.3;
6 set representation:HeadControlRequest controlType =
    direct; pan = -1; tilt = -0.3;
7 set representation:HeadControlRequest controlType =
    direct; pan = 0; tilt = 0.3;
8 set representation:HeadControlRequest controlType =
    direct; pan = 1; tilt = 0.3;
9 set representation:HeadControlRequest controlType =
    direct; pan = -1; tilt = 0.3;
```

```
10 set representation:CameraCalibration
     cameraLowerTiltCorrection = -0.026;
     cameraLowerRollCorrection = -0.018;
     cameraLowerPanCorrection = -0.003;
     cameraUpperTiltCorrection = -0.03;
     cameraUpperRollCorrection = -0.002;
     cameraUpperPanCorrection = 0.007; bodyTiltCorrection =
     0.08; bodyRollCorrection = -0.018;
     bodyTranslationCorrection = {x = 0; y = 0; z = 0;};
     upper2lowerRotation = {x = 0; y = -0.69289; z = 0;};
     upper2lowerTranslation = {x = 8; y = 0; z = 45.9;};
     colorTemperature = default;
```

Als erstes muss dann der Roboter in den richtigen Stand versetzt werden, indem man in der Zeile 2 einmal Enter drückt und dadurch den entsprechenden MotionRequest setzt. Dann muss einmal die Zeile `set representation:RobotPose rotation = 1.602; translation = {x = 0; y = -3000;}; validity = 0.2;` ausgeführt werden, damit der Roboter initial weiß wo er steht. Erst wenn das geschehen ist, startet die eigentliche Kalibrierung, bei der die oben genannten Parameter angepasst werden.

Man beginnt immer mit der oberen Kamera und dem Blick nach vorne. Diese Einstellung wird dann für die obere Kamera optimal kalibriert. Danach stellt man den Kopf nach links beziehungsweise rechts, um darüber die Body Parameter einzustellen. Als Faustformel gilt dabei, dass die Summe der beiden Parameter `cameraUpper`*`XX`* und `body`*`XX`* immer gleich bleiben muss. Wenn nun die obere Kamera in allen Richtungen optimal kalibriert ist, wird die untere Kamera kalibriert. Dabei kann es in seltenen Fällen passieren, dass die unteren Parameter nicht ausreichen um die Kamera optimal zu kalibrieren. Dann muss dies über die Body Parameter passieren. Dabei darf die obere Kamera nicht dekalibriert werden, da sonst alle Parameter noch einmal angepasst werden müssen.

14.2.3 Kalibrierungsbeispiele



Die Parameter der `CameraCalibtration` waren in diesem Beispiel von einer früheren Kalibrierung wie folgt gesetzt:

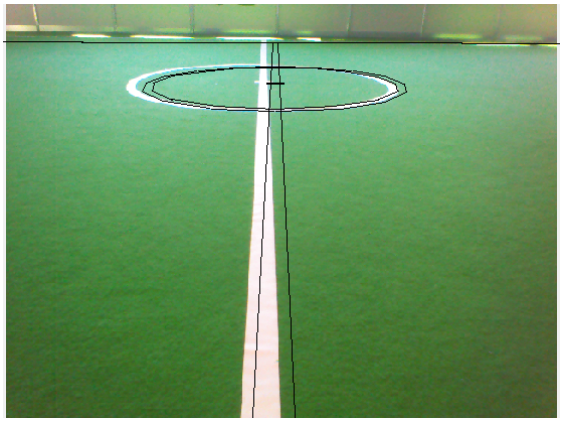
```
cameraLowerTiltCorrection = -0.026;
cameraLowerRollCorrection = -0.018;
cameraLowerPanCorrection = -0.003;
cameraUpperTiltCorrection = -0.03;
cameraUpperRollCorrection = -0.002;
cameraUpperPanCorrection = 0.007;
bodyTiltCorrection = 0.08;
bodyRollCorrection = -0.018;
```

Links sind die Bilder der oberen und unteren Kamera nach dem initialen Setzen der RobotPose. Wie an den schwarzen Linien zu erkennen ist, liegen diese nicht genau über den weißen Spielfeldlinien. Dementsprechend sind die Kamera-Matrizen noch nicht optimal kalibriert. Die schwarzen Mittellinie ist zwar zwischen den Roboterfüßen mittig, geht aber bei größerer Entfernung immer weiter nach rechts. Dies spricht für eine leichte Rotation des Roboters, da beide Bildern ungefähr gleich rotiert sind.

Hier wurde nun die RobotPose angepasst. Der Parameter `rotation` beträgt nun 1.549 statt den vorherigen 1.602. Beim oberen Bild ist nun noch eine parallele Verschiebung der Matrix nach rechts. Außerdem ist die Seitenlinie rechts weit über der Feldlinie und links ein wenig. Als nächstes wird nun der Kopf nach links und nach recht bewegt um auch dort die Verschiebungen zu analysieren. Die untere Kamera wird erst zum Schluß kalibriert.

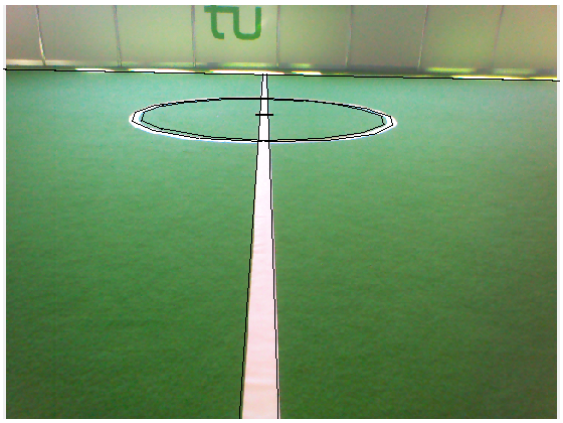
Das obere Bild zeigt die linke Blickrichtung. Die Matrix ist an der rechten Ecke nach oben gezogen. Bei der rechten Blickrichtung ist sie unter der Feldlinie. Dementsprechend muss der Parameter Roll angepasst werden und, wie beim Blick nach vorne festgestellt, muss der Tilt ein wenig nach unten korrigiert werden.



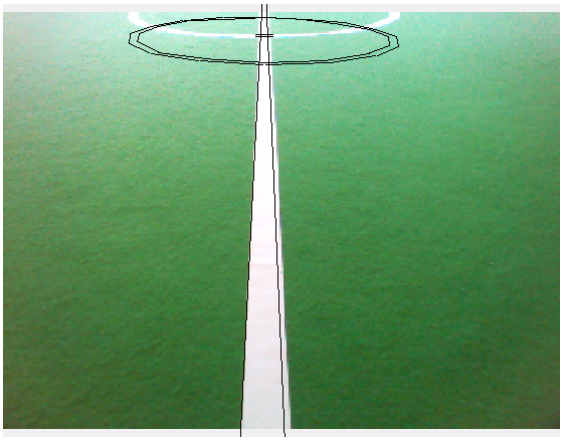


Der Tilt wurde von -0.03 auf -0.038 geändert. Außerdem wurde die `bodyRollCorrection` um 0.018 erhöht und die `cameraUpperRollCorretion` um den gleichen Wert verringert. Dies verändert den Blick nach vorne nicht stark, dafür aber den Blick nach rechts und links.

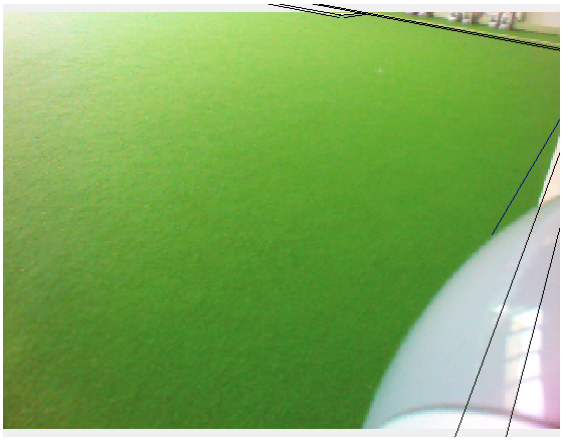


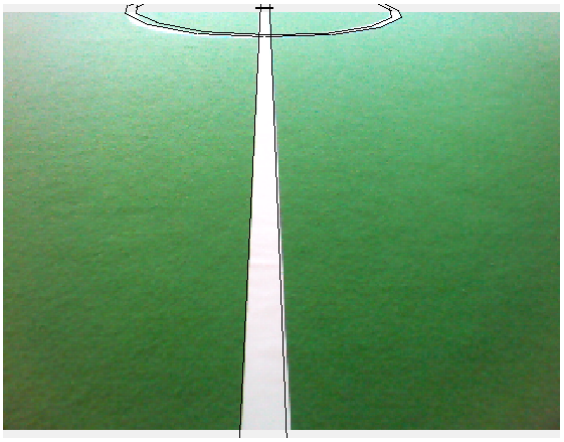


Anschließend wurde die `cameraUpperPanCorrection` auf -0.02 verringert und die `cameraUpperRollCorrection` von -0.02 auf -0.01 erhöht. Die obere Kamera ist damit kalibriert. Nun wird noch die untere Kamera angepasst.



Die untere Kamera-Matrix ist nach unten verschoben. Dementsprechend muss der Parameter `cameraLowerTiltCorrection` angepasst werden.





Die `cameraLowerTiltCorrection` wurde von -0.026 auf 0.026 angepasst. Somit ist auch die untere Kamera kalibriert.

14.2.4 Calibration-Helper

Abschließend lässt sich sagen, dass die Kamera-Kalibrierung durchaus wichtig ist, da die Roboter mit komplett fehlkalibrierten Kameras nicht sinnvoll spielen, aber gleichzeitig ist es auch eine mühsame und immer wiederkehrende Aufgabe. Allein nach einem Neustart sind die Roboter bereits wieder leicht dekalibriert und nach wenigen Spielen ist eine erneute Kalibrierung notwendig. Wenn die Robotor kalibriert werden, ist es wichtig, darauf zu achten, dass erst die Joints und danach erst die Kameras kalibriert werden, da es bei der Joint-Kalibrierung wieder zu einer Dekalibrierung der Kameras kommen kann. In Folge

des erhöhten und immer wiederkehrenden Aufwands ist auf dem Robocup ein Tool mit dem Namen `Calibration-Helper` (siehe Abbildung 14.6) entstanden. Das Tool wurde in Java geschrieben und erleichtert die Arbeit der Kalibrierung, indem sich die einzustellenden Parameter komfortabel mit Schiebereglern verändern lassen. Während die Parameter verändert werden, wird im Hintergrund der benötigte Code (siehe Listing 14.1) in die Zwischenablage kopiert. Dies bedeutet, dass man in den Simulator den fertigen Code nur einfügen muß. So entfällt das langwierige Suchen und manuelle Ersetzen von Parametern direkt im Code des Simulators.

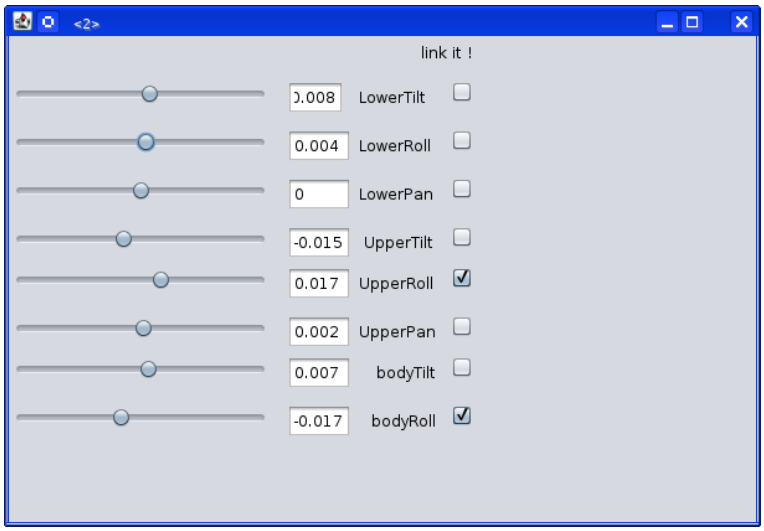


Abbildung 14.6: Calibration-Helper

Teil V

Fazit

bearbeitet von: Michael Feininger

Mit dem RoboCup 2013 war auch dieses Jahr die Herausforderung gegeben, sich in verschiedenen Forschungsbereichen mit anderen Nationen zu messen und die Ziele der Projektgruppe zu realisieren. Dabei wurden unterschiedliche Bereiche der Robotik erarbeitet und vorgestellt. Die Vorstellung in den einzelnen RoboCup-Wettkämpfen ermöglichte einen Überblick über die Entwicklung und Forschung auf unterschiedlicher Weise.

Die Entwicklung eines Debugging-Tools ermöglichte eine sehr detailreiche und präzise Auswertung, die bei der Analyse und Behebung unterschiedlicher Schwächen hilfreich war.

Durch die Vergrößerung des Spielfeldes in diesem Jahr mussten einige Verhalten modifiziert oder neu geschrieben werden und bildeten das Grundgerüst für neue Spielstrategien. Dadurch wurden dieses Jahr zwei Taktiken ausgearbeitet sowohl die Triangletactic und die Raumaufteilung, um ein möglichst gutes Ergebnis zu erzielen wurde jedoch das Triangleverhalten aufgegeben. Zusätzlich wurden noch die Verhalten des Torwartes sowie das Verhalten beim Elfmeter erneuert. Besonders das Verhalten des Torwartes erwies sich als stabil und konsequent in der Handlung. In der DropIn-Challenge glänzte der Torhüter auch mit einer Parade und mit dem Tor eines unserer Spieler war der zweite Platz in dieser Challenge gesichert. Die Passing-Challenge zeigte sich ebenfalls stabil, doch aufgrund der strapazierten Roboter verfehlten die Pässe ihr Ziel um wenige Zentimeter und brachten ein dafür eher unbefriedigten dritten Platz. Bei der Open-Challenge der Unterstützer App für den Schiedsrichter, dem SmartRef, gab es von mehreren Teams positive Resonanz und ebenfalls eine gute vierte Platzierung in der Bewertung. Das Elfmeterverhalten kam im Wettbewerb nicht zum Einsatz. Insgesamt haben die umgesetzten Verhalten zum Viertelfinale in dem SPL-Turnier geführt und einen dritten Platz in den Challenges, hinter den Finalisten B-Human aus Bremen und HTWK aus Leipzig, eingebracht.

Die kontinuierliche Integration bot durch die gesamte Implementierungsphase hindurch ein lauffähiges Projekt. Zu jedem Zeitpunkt war das Projekt kompilier- und ausführbar. Die Entwickler konnten so ihre Fortschritte fortlaufend in das Projekt integrieren. Durch das neu portierte Framework konnte die Hardware besser verwendet werden. Die Portierung auf das neue Framework brachte einige Anpassungen mit sich und verzögerte etwas allgemeine die Weiterentwicklung. Anschließend mussten die Kameras neben dem Lauf (Join) kalibriert werden, sorgten so für Stabilität und zügige Entscheidungen auf dem Spielfeld.

Zusätzlich zu den Zielen wurde ein DeployerTool (NaoDeployer) entwickelt, dass der Übersichtlichkeit dient und ein schnelleres Aufsetzen der Roboter ermöglicht. Schlussendlich begeisterte die Kombination von Wissenschaft, Technik, Sport und Unterhaltung die komplette Projektgruppe und aufbauend auf den Ergebnissen überlegen sich einige auch die Abschlussarbeit im Institut für Roboterforschung zu schreiben.

Teil VI

Anhang

Abbildungsverzeichnis

Literaturverzeichnis

[1] *Aldebaran Robotics*. http://www.aldebaran-robotics.com.

[2] RoboCup Technical Committee: *RoboCup Standard Platform League (Nao) Rule Book - (2013 rules, as of March 5, 2013)*. www.tzi.de/spl/pub/Website/Downloads/Rules2013.pdf, 2013.

[3] *Java Docking Frames Dokumentation*. http://dock.javaforge.com/doc.html.

[4] *JFreeChart Dokumentation*. http://www.jfree.org/jfreechart/api.html.

[5] Ullenboom, Christian: *Java ist auch eine Insel*. http://openbook.galileodesign.de/javainsel5/index.htm, 2006.

[6] *VLCJ Dokumentation*. http://caprica.github.io/vlcj/javadoc/2.1.0/.

[7] *SuperCSV Dokumentation*. http://supercsv.sourceforge.net/index.html.

[8] *Java Swing TreeTable*. http://hameister.org/JavaSwingTreeTable.html.

[9] Röfer, Thomas, Tim Laue, Judith Müller, Michel Bartsch, Malte Jonas Batram, Arne Böckmann, Nico Lehmann, Florian Maass, Thomas Münder, Marcel Steinbeck, Andreas Stolpmann, Simon Taddiken, Robin Wieschendorf und Danny Zitzmann: *B-Human Team Report and Code Release 2012*, 2012. Only available online: http://www.b-human.de/wp-content/uploads/2012/11/CodeRelease2012.pdf.

[10] Fowler, Martin: *Continuous Integration*. http://www.martinfowler.com/articles/continuousIntegration.html, 2006.

[11] Fowler, Martin: *About Me*. http://martinfowler.com/aboutMe.html, 2006.

[12] *Nao Devils Homepage*. http://www.nao-devils.de.

[13] Berlin, Jan Hendrik, Bastian Böhm, Sebastian Drywa, Elena Erdmann, Michael Feininger, Florian Gürster, Diana Howey, Christian Kroll, Stefan Papon, Bianca Patro, Heinrich Pettenpohl und Yuri Struszczynski: *Zwischenbericht der Projektgruppe 571*, 2013.

[14] *TeamCity Download*. http://www.jetbrains.com/teamcity/download.

[15] *TeamCity Dokumentation*. http://www.jetbrains.com/teamcity/documentation/index.jsp.

[16] *SSH.NET Library*. http://sshnet.codeplex.com/, 2013.

[17] Berlin, Jan Hendrik: *github of Naodeployer*. https://github.com/NaoDevils/NaoDeployer, 2013.

[18] Holzmann, Günther, Heinz Meyer und Georg Schumpich: *Technische Mechanik Kinematik und Kinetik*. Springer Vieweg, 11. Aufl. Auflage, 2012.

[19] *Holmes-Wiki*. http://holmes.it.irf.tu-dortmund.de/wiki/doku.php?

[20] Beiersmann, Stefan: *Android erreicht 70% Marktanteil in Europa*. http://www.zdnet.de/88160639/android-erreicht-70-prozent-marktanteil-in-europa/, 2013.

[21] Google: *Android developers*. http://developer.android.com/reference/, Juni 2013.

[22] Gramlich, Nicolas: *Andengine*. https://github.com/nicolasgramlich/AndEngine/, 2013.

[23] Bartsch, M., M. Steinbeck, R. Wieschendorf, and T. Röfer: *Gamecontroller2*. https://github.com/bhuman/GameController, 2012.

[24] Catto, Erin: *Box2d*. http://box2d.org/, 2013.

[25] Gramlich, Nicolas: *Andengine physics box2d extension*. https://github.com/nicolasgramlich/AndEnginePhysicsBox2DExtension, 2013.

[26] Group, Khronos: *Opengl es 2.x - for programmable hardware*. http://www.khronos.org/opengles/2_X/, Mai 2013.

[27] Google: *Android dashboards*. http://developer.android.com/about/dashboards/index.html, Juli 2013.

[28] RoboCup SPL Technical Committee: *Robocup 2013 - spl - results*. http://www.tzi.de/spl/bin/view/Website/Results2013, 2013.

[29] *Gnu gplv2 + classpath exception for openjdk*. http://openjdk.java.net/legal/gplv2+ce.html.

[30] Topic, Dalibor: *Retiring the dlj*. http://robilad.livejournal.com/90792.html, 2011.

[31] *Java se downloads*. http://www.oracle.com/technetwork/java/javase/downloads/index.html.

[32] *Hypersql database*. http://hsqldb.org.

[33] *Xabsl webseite*. http://www.xabsl.de/.

[34] Christian, Deutsch: *Pfadplanung für einen autonomen mobilen roboter*. Diplomarbeit, Technische Universität Graz, 2004.

[35] Czarnetzki, Stefan, Sören Kerner, Oliver Urbann, Matthias Hofmann, Sven Stumm, and Ingmar Schwarz: *Nao devils dortmund team report 2010*. Technical report, Robotics Research Institute, TU Dortmund University, 2010.

[36] Tasse, Stefan, Sören Kerner, Oliver Urbann, Matthias Hofmann, and Ingmar Schwarz: *Nao devils dortmund team report 2011.* Technical report, Robotics Research Institute, TU Dortmund University, 2011.

[37] Schroeder, J.: *AndEngine for Android Game Development Cookbook.* Packt Publishing, Limited, 2013.

[38] Rogers, R.A.: *Learning Android Game Programming: A Hands-On Guide to Building Your First Android Game.* Learning. Pearson Education, 2011.

[39] Becker, A. and M. Pant: *Android: Grundlagen und Programmierung.* dpunkt-Verlag, 2009.

[40] Wilson, Jonathan: *Revolutionen auf dem Rasen - Eine Geschichte der Fußballtaktik.* Verlag die Werkstatt, 3. aufl. edition, 2011.

[41] RoboCup SPL Technical Committee: *Technical challenges for the robocup 2013 standard platform league competition (march 29, 2013).* http://www.tzi.de/spl/pub/Website/Downloads/Challenges2013.pdf, 2013.